

PYTHON AU LYCÉE

TOME 2

ARNAUD BODIN

ALGORITHMES ET PROGRAMMATION



Python au lycée – tome 2

Informatique et ordinateur

E. Dijkstra a dit que « l'informatique est autant la science des ordinateurs que l'astronomie est la science des télescopes ». Une partie fondamentale de l'informatique est en effet la science des algorithmes : comment résoudre un problème le plus efficacement possible. Un algorithme étant une suite d'instructions théoriques indépendantes du langage et de la machine utilisée. Mais il faut comprendre le « autant » de façon positive : les astronomes ont besoin de télescopes performants autant que les informaticiens d'ordinateurs puissants. Pour programmer intelligemment il faut donc bien connaître sa machine, ses limitations mais aussi le langage utilisé.

Python

Le but de ce second volume est d'approfondir notre connaissance de Python. Tu vas écrire des programmes de plus en plus compliqués et résoudre à la machine des grilles de sudoku, les calculs du « compte est bon » et la recherche du « mot le plus long ». Tu vas aussi programmer de belles images : des automates cellulaires, du traitement d'images, des surfaces, des dessins en perspective et de nombreuses fractales. Tu vas aussi découvrir de nouveaux algorithmes pour trier, pour calculer en parallèle, pour résoudre des équations. Parmi les nouveaux outils que tu vas découvrir il y aura les algorithmes récursifs, la programmation objet, les dictionnaires.

Mathématiques

Contrairement au premier tome on ne se limite plus aux mathématiques du niveau seconde. Voici les chapitres abordés de niveau première et terminale : suites, dérivées, intégration, nombres complexes, logarithme, exponentielle, matrices.

L'intégralité des codes Python des activités ainsi que tous les fichiers sources sont sur la page *GitHub* d'Exo7 : « [GitHub : Python au lycée](#) ».

Les vidéos des notions de base et des activités du premier tome sont accessibles depuis la chaîne *Youtube* : « [Youtube : Python au lycée](#) ».

Sommaire

I	Mathématiques avec informatique	1
1	Suites arithmétiques – Suites géométriques	2
2	Nombres complexes I	9
3	Nombres complexes II	15
4	Dérivée – Zéros de fonctions	22
5	Exponentielle	31
6	Logarithme	37
7	Intégrale	52
II	Informatique avec mathématiques	57
8	Programmation objet	58
9	Mouvement de particules	70
10	Algorithmes récursifs	79
11	Tri – Complexité	97
12	Calculs en parallèle	107
III	Projets	122
13	Automates	123
14	Cryptographie	129
15	Le compte est bon	136
16	Le mot le plus long	142
17	Images et matrices	150

18	Ensemble de Mandelbrot	162
19	Images 3D	169
20	Sudoku	189
21	Fractale de Lyapunov	202
22	Big data I	212
23	Big data II	224
IV	Guides	244
24	Guide de survie Python	245
25	Principales fonctions	256
26	Notes et références	271
	Index	

Résumé des activités

La plupart des activités sont indépendantes les unes des autres.

Tu peux commencer par celles qui te font le plus envie !

Suites arithmétiques – Suites géométriques

Tu vas manipuler deux types de suites fondamentales : les suites arithmétiques et les suites géométriques.

Nombres complexes I

Nous allons faire des calculs avec les nombres complexes. Ce sera facile car Python sait les manipuler.

Nombres complexes II

On poursuit l'exploration des nombres complexes en se concentrant sur la forme module/argument.

Dérivée – Zéros de fonctions

Nous étudions les fonctions : le calcul de la dérivée d'une fonction, le tracé du graphe et de tangentes, et enfin la recherche des valeurs où la fonction s'annule.

Intégrale

Nous allons étudier différentes techniques pour calculer des valeurs approchées d'intégrales.

Exponentielle

L'exponentielle joue un rôle important dans la vie de tous les jours : elle permet de modéliser la vitesse de refroidissement de votre café, de calculer la croissance d'une population ou de calculer la performance d'un algorithme.

Logarithme

Le logarithme est une fonction aussi importante que l'exponentielle. C'est le logarithme qui donne l'ordre de grandeur de certaines quantités physiques, par exemple la puissance d'un séisme ou celle d'un son.

Programmation objet

Avec Python tout est objet : un entier, une chaîne, une liste, une fonction... Nous allons voir comment définir nos propres objets.

Mouvement de particules

Tu vas simuler le mouvement d'une particule soumise à différentes forces, comme la gravité ou des frottements. Tu appliqueras ceci afin de simuler le mouvement des planètes autour du Soleil. Cette activité utilise la programmation objet.

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.

Tri – Complexité

Ordonner les éléments d'une liste est une activité essentielle en informatique. Par exemple une fois qu'une liste est triée, il est très facile de chercher si elle contient tel ou tel élément. Par définition un algorithme renvoie toujours le résultat attendu, mais certains algorithmes sont plus rapides que d'autres ! Cette efficacité est mesurée par la notion de complexité.

Calculs en parallèle

Comment profiter d'avoir plusieurs processeurs (ou plusieurs cœurs dans chaque processeur) pour calculer plus vite ? C'est simple il s'agit de partager les tâches afin que tout le monde travaille en même temps, puis de regrouper les résultats. Dans la pratique ce n'est pas si facile.

Automates

Tu vas programmer des automates cellulaires, qui à partir de règles simples, produisent des comportements amusants.

Cryptographie

Tu vas jouer le rôle d'un espion qui intercepte des messages secrets et tente de les décrypter.

Images et matrices

Le traitement des images est très utile, par exemple pour les agrandir ou bien les tourner. Nous allons aussi voir comment rendre une image plus floue, mais aussi plus nette ! Tout cela à l'aide des matrices.

Le compte est bon

Qui n'a jamais rêvé d'épater sa grand-mère en gagnant à tous les coups au jeu « Des chiffres et des lettres » ? Une partie du jeu est « Le compte est bon » dans lequel il faut atteindre un total à partir de chiffres donnés et des quatre opérations élémentaires. Pour ce jeu les ordinateurs sont plus rapides que les humains, il ne te reste plus qu'à écrire le programme !

Le mot le plus long

La seconde partie du jeu « Des chiffres et des lettres » est le « Le mot le plus long ». Il s'agit simplement de trouver le mot le plus grand à partir d'un tirage de lettres. Pour savoir si un mot est valide on va utiliser une longue liste des mots français.

Ensemble de Mandelbrot

Tu vas découvrir un univers encore plus passionnant qu'*Harry Potter* : l'ensemble de Mandelbrot. C'est une fractale, c'est-à-dire que lorsque l'on zoome sur certaines parties de l'ensemble, on retrouve une image similaire à l'ensemble de départ. On découvrira aussi les ensembles de Julia.

Images 3D

Comment dessiner des objets dans l'espace et comment les représenter sur un plan ?

Sudoku

Tu vas programmer un algorithme qui complète entièrement une grille de sudoku. La méthode utilisée est la recherche par l'algorithme du « retour en arrière ».

Fractale de Lyapunov

Nous allons étudier des suites dont le comportement peut être chaotique. La fonction logarithme nous aidera à déterminer le caractère stable ou instable de la suite. Avec beaucoup de calculs et de patience nous tracerons des fractales très différentes de l'ensemble de Mandelbrot : les fractales de Lyapunov.

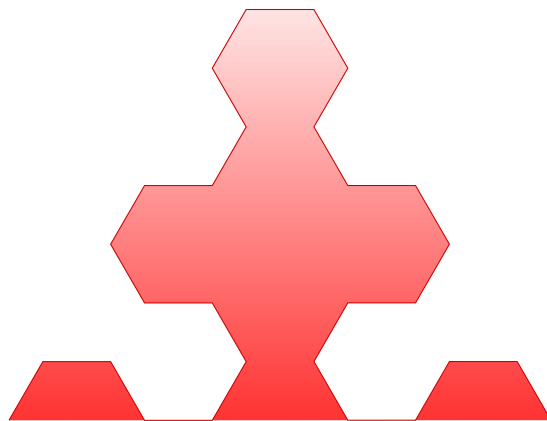
Big data I

Big data, intelligence artificielle, *deep learning*, réseau de neurones, *machine learning*. . . plein de mots compliqués ! Le but commun est de faire exécuter à un ordinateur des tâches de plus en plus complexes : *choisir* (par exemple trouver un bon élément parmi des milliards selon plusieurs critères), *décider* (séparer des photos de chats de photos de voitures), *prévoir* (un malade a de la fièvre et le nez qui coule, quelle maladie est la plus probable ?). Dans cette première partie on va utiliser des outils classiques de statistique et de probabilité pour résoudre des problèmes amusants.

Big data II

L'essor des *big-data* et de l'intelligence artificielle est dû à l'apparition de nouveaux algorithmes adaptés à la résolution de problèmes complexes : reconnaissance d'images, comportement des électeurs, conduite autonome de voitures. . . Dans cette seconde partie tu vas programmer quelques algorithmes emblématiques et innovants.

PREMIÈRE PARTIE



MATHÉMATIQUES AVEC INFORMATIQUE

Suites arithmétiques – Suites géométriques

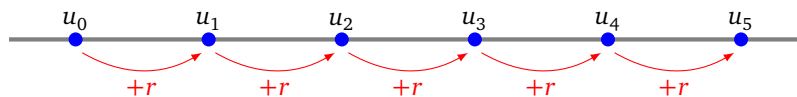
Chapitre

1

Tu vas manipuler deux types de suites fondamentales : les suites arithmétiques et les suites géométriques.

Cours 1 (Suites arithmétiques).

Une suite arithmétique est une suite telle que la différence entre deux termes consécutifs ait toujours la même valeur.



1. **Définition.** Une suite $(u_n)_{n \in \mathbb{N}}$ est une **suite arithmétique** de **raison** r si on a $u_{n+1} = u_n + r$ pour tout $n \geq 0$.
2. **Formule de récurrence.** Une suite arithmétique est donc entièrement définie par son premier terme u_0 et sa raison r :

terme initial u_0 et
formule de récurrence $u_{n+1} = u_n + r$

3. **Formule directe.** On calcule u_n directement par la formule :

$$u_n = nr + u_0$$

4. **Exemple.**

7 10 13 16 19 ...

C'est la suite arithmétique de terme initial $u_0 = 7$ et de raison $r = 3$. La formule directe est $u_n = 3n + 7$.

5. **Somme.** La somme des termes de u_0 jusqu'à u_n est donnée par la formule :

$$S_n = u_0 + u_1 + u_2 + \cdots + u_n = (n+1)u_0 + \frac{n(n+1)}{2}r$$

Activité 1 (Suites arithmétiques).

Objectifs : programmer les différentes formules autour des suites arithmétiques.

1. Programme une fonction `arithmetique_1(n,u0,r)` qui renvoie le terme de rang n de la suite arithmétique définie par le terme initial u_0 et la raison r , en utilisant la formule de récurrence. Quel est le terme u_{100} de la suite arithmétique définie par $u_0 = 13$ et $r = 5$?
2. Programme une fonction `arithmetique_2(n,u0,r)` qui fait la même chose mais en utilisant cette

fois la formule directe.

3. Programme une fonction `liste_arithmetique(n,u0,r)` qui renvoie la liste des termes $[u_0, u_1, u_2, \dots, u_n]$.
4. Programme une fonction `est_arithmetique(liste)` qui teste si les termes $[u_0, u_1, u_2, \dots, u_n]$ de la liste donnée forment le début d'une suite arithmétique.

Indications.

- Le programme renvoie `True` ou `False`.
 - On suppose que la liste contient au moins deux éléments.
 - Si la liste est constituée des premiers termes d'une suite arithmétique alors, le terme initial est u_0 et la raison est $r = u_1 - u_0$. Et on doit avoir $u_{n+1} - u_n = r$ pour tout n . Tu peux alors utiliser la question précédente.
 - Exemple : avec $[3, 5, 7, 10]$ la fonction renvoie « Faux ».
5. Programme une fonction `somme_arithmetique_1(n,u0,r)` qui calcule, en additionnant les éléments, la somme des termes de rang 0 à n d'une suite arithmétique de terme initial u_0 et de raison r . Retrouve le même résultat par une fonction `somme_arithmetique_2(n,u0,r)` qui utilise la formule de la somme donnée dans le cours ci-dessus.

Combien vaut la somme :

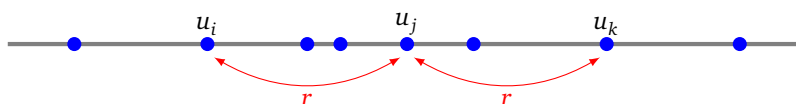
$$2 + 4 + 6 + 8 + \dots + 1000 ?$$

Activité 2 (Trois termes d'une suite arithmétique).

Objectifs : déterminer si dans une liste donnée il existe trois termes d'une suite arithmétique.

On te donne une liste ordonnée $[u_0, u_1, u_2, \dots, u_n]$. Tu dois déterminer si dans cette liste on peut trouver trois termes u_i, u_j, u_k qui font partie d'une suite arithmétique. Autrement dit, tels que :

$$u_i = u_j - r \quad u_k = u_j + r \quad \text{pour un certain } r.$$



Par exemple dans la liste :

$$[10, 11, 13, 17, 19, 20, 23, 29, 31]$$

les trois termes $u_i = 11$, $u_j = 17$, $u_k = 23$ sont en progression arithmétique, de raison $r = 6$.

Programme l'algorithme ci-dessous en une fonction `chercher_arithmetique(u)` qui à partir d'une liste de termes u renvoie trois termes en progression arithmétique (ou `None` s'il n'y en a pas).

Le principe de l'algorithme est le suivant. Pour chaque élément u_j de la suite (qui va jouer le rôle du potentiel élément central) :

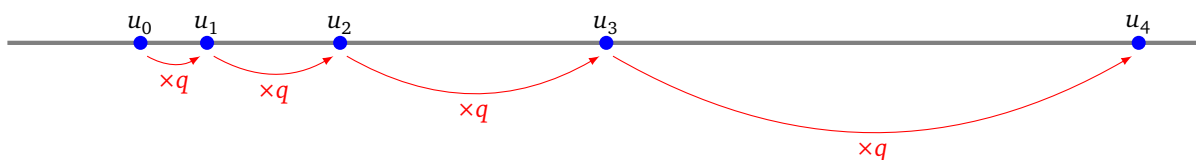
- On cherche un élément u_i de rang i plus petit que j et un élément u_k de rang k plus grand que j avec $u_j - u_i = u_k - u_j$ (on aura alors $u_j = u_i + r$ puis $u_k = u_j + r$). Si on a cette égalité alors c'est gagné !
- Si on n'a pas cette égalité alors on prend un i plus petit ou bien un k plus grand.

Algorithme.

- — Entrée : une liste de termes $[u_0, u_1, \dots, u_n]$ ordonnée.
- Sortie : trois termes en progression arithmétique (ou rien s'il n'y en pas).
- Pour j parcourant les indices de 1 à $n - 1$:
 - Poser $i = j - 1, k = j + 1$.
 - Tant que $i \geq 0$ et $k \leq n$:
 - Si $u_j - u_i = u_k - u_j$ renvoyer le triplet u_i, u_j, u_k (qui forme une progression arithmétique).
Le programme s'arrête là avec succès.
 - Si $u_j - u_i < u_k - u_j$ alors faire $i \leftarrow i - 1$.
 - Si $u_j - u_i > u_k - u_j$ alors faire $k \leftarrow k + 1$.
- Lorsque la boucle « pour » se termine sans avoir obtenu de triplet, c'est qu'il n'y en a pas.

Cours 2 (Suites géométriques).

Pour une suite géométrique le quotient entre deux termes consécutifs est toujours le même.



1. **Définition.** Une suite $(u_n)_{n \in \mathbb{N}}$ est une **suite géométrique** de **raison** q si on a $u_{n+1} = qu_n$ pour tout $n \geq 0$.
2. **Formule de récurrence.** Une suite géométrique est donc entièrement définie par son premier terme u_0 et sa raison q :

terme initial u_0 et
formule de récurrence $u_{n+1} = qu_n$

3. **Formule directe.** On calcule u_n directement par la formule :

$$u_n = u_0 \cdot q^n$$

4. **Exemple.**

2 6 18 54 162 ...

est le début de la suite géométrique de terme initial $u_0 = 2$, de raison $q = 3$. La formule directe est $u_n = 2 \times 3^n$.

5. **Somme.** La somme des termes de u_0 jusqu'à u_n (pour $q \neq 1$) est donnée par la formule :

$$S_n = u_0 + u_1 + u_2 + \dots + u_n = u_0 \times \frac{1 - q^{n+1}}{1 - q}$$

que l'on mémorise par :

$$\text{somme suite géométrique} = \text{terme initial} \times \frac{1 - \text{raison}^{\text{nombre de termes}}}{1 - \text{raison}}$$

Activité 3 (Suites géométriques).

Objectifs : refaire la première activité sur les suites arithmétiques, mais cette fois pour les suites géométriques.

1. Programme une fonction `geometrique_1(n,u0,q)` qui renvoie le terme de rang n de la suite géométrique définie par le terme initial u_0 et la raison q , en utilisant la formule de récurrence. Quel est le terme u_{10} de la suite géométrique définie par $u_0 = 13$ et $r = 5$?
2. Programme une fonction `geometrique_2(n,u0,q)` qui fait la même chose mais en utilisant cette fois la formule directe.
3. Programme une fonction `liste_geometrique(n,u0,q)` qui renvoie la liste des termes $[u_0, u_1, u_2, \dots, u_n]$.
4. Programme une fonction `est_geometrique(liste)` qui teste si les termes $[u_0, u_1, u_2, \dots, u_n]$ de la liste donnée forment le début d'une suite géométrique.

Indications. Si la liste est constituée des premiers termes d'une suite géométrique alors, $\frac{u_{n+1}}{u_n} = \frac{u_1}{u_0}$ pour tout n . Utilise la question précédente.

5. Programme une fonction `somme_geometrique_1(n,u0,q)` qui calcule, en additionnant les éléments, la somme des termes de rang 0 à n d'une suite géométrique de terme initial u_0 et de raison q . Retrouve le même résultat par une fonction `somme_geometrique_2(n,u0,q)` qui utilise la formule de la somme donnée dans le cours ci-dessus.

Vers quelle valeur a l'air de tendre la somme :

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots + \frac{1}{2^n}$$

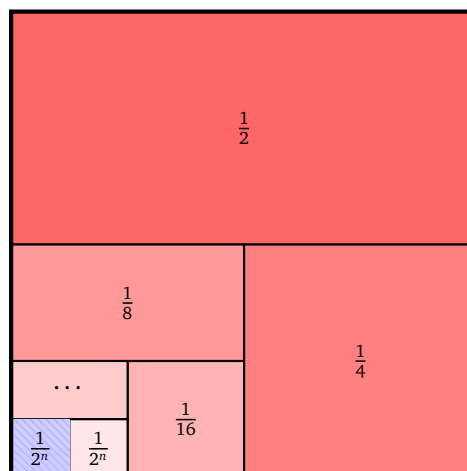
lorsque n tend vers l'infini ?

Activité 4 (Tracer la somme d'une suite géométrique).

Objectifs : illustrer géométriquement la formule de la somme d'une suite géométrique.

Voici un découpage d'un carré de côté 1 qui illustre la formule :

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots + \frac{1}{2^n} = 1 - \frac{1}{2^n}$$



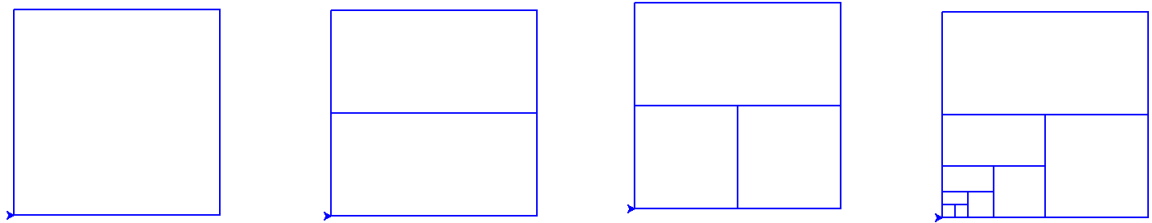
1. Programme une fonction `affiche_un_carre(longueur)` qui affiche un carré de la longueur donnée. Utilise la tortue accessible depuis le module `turtle`.

2. Programme une fonction `affiche_un_rectangle(longueur)` qui trace un rectangle de hauteur la moitié de sa longueur. Il coupe le carré précédent en deux parties égales.
3. Programme une fonction `affiche_les_carres(n)` qui construit notre figure.

Indications.

- Par exemple, on commence par tracer un carré de longueur 256,
- on trace un rectangle qui coupe le carré en deux,
- puis on trace un carré de longueur 128,
- puis on le découpe en deux, etc.

De gauche à droite : le carré initial ; le carré coupé en deux rectangles ; un petit carré ; un découpage itéré.



Preuves de la formule.

On considère la suite :

$$\frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{8} \quad \frac{1}{16} \quad \dots \quad \frac{1}{2^n} \quad \dots$$

C'est la suite géométrique (u_n) de terme initial $u_0 = \frac{1}{2}$ et de raison $q = \frac{1}{2}$.

Preuve par le dessin.

Le grand carré a pour aire 1, l'aire totale des zones rouges est $\frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^n}$. La zone hachurée bleue a pour aire $\frac{1}{2^n}$. Les zones rouges et bleues recouvrent tout le carré, donc leur aire totale vaut 1. Ce qui prouve la formule annoncée :

$$\underbrace{\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots + \frac{1}{2^n}}_{\text{aire rouge}} + \underbrace{\frac{1}{2^n}}_{\text{aire bleue}} = \underbrace{1}_{\text{aire du grand carré}}$$

Preuve par le calcul.

La formule pour la somme est

$$S_{n-1} = u_0 + u_1 + u_2 + \dots + u_{n-1} = u_0 \times \frac{1 - q^n}{1 - q}$$

(attention il y a bien n termes dans la somme) et donc ici :

$$S_{n-1} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots + \frac{1}{2^n} = \frac{1}{2} \times \frac{1 - \frac{1}{2^n}}{1 - \frac{1}{2}} = 1 - \frac{1}{2^n}$$

Activité 5 (Meilleure suite arithmétique).

Objectifs : on te donne une liste ordonnée, tu dois trouver la suite arithmétique qui approche le mieux possible cette liste.

Qu'est ce que la meilleure suite arithmétique qui approche une liste de nombres donnés ? Par exemple pour la liste $[3, 6, 9, 11]$, on a envie de l'approcher par la progression arithmétique $[3, 6, 9, 12]$.

On nous donne donc des termes $v_0, v_1, \dots, v_n$ (ordonnés du plus petit au plus grand). On va chercher

une progression arithmétique $u_0, u_1, \dots, u_n$ telle que

$$d = |v_0 - u_0| + |v_1 - u_1| + |v_2 - u_2| + \dots + |v_n - u_n|$$

soit le plus petit possible.

On appelle d la **distance** entre $[v_0, v_1, \dots, v_n]$ et $[u_0, u_1, \dots, u_n]$. Pour l'exemple donné, $[3, 6, 9, 11]$ approchée par $[3, 6, 9, 12]$, la distance vaut 1.

1. **Distance.** Programme une fonction `distance(u, v)` qui calcule la distance

$$d = |v_0 - u_0| + |v_1 - u_1| + |v_2 - u_2| + \dots + |v_n - u_n|$$

entre deux listes $u = [u_0, u_1, \dots, u_n]$ et $v = [v_0, v_1, \dots, v_n]$.

2. **Meilleure constante.** On nous donne une liste $w = [w_0, w_1, \dots, w_n]$, on cherche une constante m qui approche au mieux toutes les valeurs de la liste, c'est-à-dire telle que

$$d = |w_0 - m| + |w_1 - m| + |w_2 - m| + \dots + |w_n - m|$$

soit le plus petit possible.

Un nombre m qui convient est simplement la médiane de la liste ! Par exemple pour $[3, 6, 9, 11]$, la médiane est $m = 7.5$ et on a

$$d = |3 - 7.5| + |6 - 7.5| + |9 - 7.5| + |11 - 7.5| = 11$$

et on ne peut pas faire moins.

Écris une fonction `calcule_médiane(liste)` qui calcule la valeur médiane des éléments d'une liste. Par définition, la moitié des valeurs est inférieure ou égale à la médiane, l'autre moitié est supérieure ou égale à la médiane. Voir le rappel de cours juste après cette activité pour ce calcul.

3. **Meilleure suite.** On va maintenant résoudre notre problème initial. On nous donne donc une liste $v = [v_0, v_1, \dots, v_n]$ et on cherche une progression arithmétique $u = [u_0, u_1, \dots, u_n]$. Pour trouver les (u_i) on doit donc trouver un terme initial u_0 et une raison r .

Méthode.

- On va d'abord trouver un r approché qui convient bien par une méthode de balayage. On cherche le meilleur r en commençant par $r = 0$ puis, par petits pas on teste jusqu'à, par exemple, $r = 2(v_1 - v_0)$.
- Pour chaque r , le terme initial u_0 qui convient est la médiane de la liste $(v_i - ir)$. (Justification : il faut minimiser la somme des $|v_i - u_i| = |v_i - ir - u_0|$; u_0 est donc la médiane des $(v_i - ir)$.)

Programme l'algorithme suivant en une fonction `balayage(v, N)` qui renvoie le terme initial u_0 et la raison r d'une suite arithmétique qui approche au mieux $v = [v_0, v_1, \dots, v_n]$. Le paramètre N correspond à la précision du balayage (plus N est grand, plus l'approximation sera bonne).

Algorithme.

- — Entrée : une liste ordonnée de termes $v = [v_0, v_1, \dots, v_n]$ et un entier N .
- Sortie : un terme initial u_0 et une raison r .
- Définir un pas $p = 2^{\frac{v_1 - v_0}{N}}$ (ce sera le pas pour le balayage de r).
- Initialiser une valeur $d_{\min}$ par une très grande valeur (par exemple $d_{\min} = 10\,000$), cette variable stockera la distance la plus petite rencontrée. Deux variables $r_{\min}$ et $u_{0,\min}$ mémoriseront les meilleurs r et u_0 trouvés.
- Poser $r = 0$.
- Pour k allant de 0 à $N + 1$:
 - Calculer u_0 la médiane de $(v_i - ir)$ (pour $0 \leq i \leq n$).
 - Définir u , la liste des premiers termes de la suite arithmétique de terme initial u_0 et de raison r (tu peux utiliser la fonction `liste_arithmetique(n, u0, r)` de la première activité).
 - Calculer la distance d entre les listes u et v .
 - Si $d < d_{\min}$ alors faire : $d_{\min} \leftarrow d$; $r_{\min} \leftarrow r$ et $u_{0,\min} \leftarrow u_0$.
 - Faire $r \leftarrow r + p$.
- Renvoyer $u_{0,\min}$ et $r_{\min}$.

Quelle est la meilleure progression arithmétique pour approcher la liste $[6, 11, 14, 20, 24, 29, 37]$?

Cours 3 (Médiane).

Par définition de la **médiane**, la moitié des valeurs sont inférieures ou égales à la médiane, l'autre moitié sont supérieures ou égales à la médiane.

Voici comment calculer la médiane. On note n la longueur de la liste, on suppose que la liste est ordonnée (du plus petit au plus grand élément).

- **Cas n impair.** La médiane est la valeur de la liste au rang $\frac{n-1}{2}$. Exemple avec `liste = [12, 12, 14, 15, 19]` :
 - la longueur de la liste est $n = 5$ (les indices vont de 0 à 4),
 - l'indice du milieu est l'indice 2,
 - la médiane est la valeur `liste[2]`, c'est donc 14.
- **Cas n pair.** La médiane est la moyenne entre la valeur de la liste au rang $\frac{n}{2} - 1$ et celle au rang $\frac{n}{2}$. Exemple avec `liste = [13, 14, 19, 20]` :
 - la longueur de la liste est $n = 4$ (les indices vont de 0 à 3),
 - les indices du milieu sont 1 et 2,
 - la médiane est la moyenne entre `liste[1]` et `liste[2]`, c'est donc $\frac{14+19}{2} = 16.5$.

Nombres complexes I

Nous allons faire des calculs avec les nombres complexes. Ce sera facile car Python sait les manipuler.

Cours 1 (Nombres complexes).

Avec Python, tu manipules les nombres complexes comme les autres nombres. La notation pour le nombre complexe i (qui vérifie $i^2 = -1$) est le symbole j (plus exactement $1j$). Par exemple, le nombre complexe $4-3i$ se note $4-3j$. Ensuite les opérations classiques s'écrivent comme d'habitude : par exemple le calcul $(1+2i)(4-i)$ s'écrit $(1+2j)*(4-1j)$ et Python renvoie $6+7j$.

- Addition $z_1 + z_2$: `z1 + z2`
- Multiplication $z_1 \cdot z_2$: `z1 * z2`
- Puissance z^n : `z1 ** n`
- Inverse $\frac{1}{z}$: `1/z`
- Partie réelle a de $z = a + ib$: `z.real` (sans parenthèses)
- Partie imaginaire b de $z = a + ib$: `z.imag` (sans parenthèses)
- Module $|z| = \sqrt{a^2 + b^2}$: `abs(z)`
- Conjugué $\bar{z} = a - ib$: `z.conjugate()`

Bien sûr Python ne fait pas toujours des calculs exacts (par exemple, lors d'une division ou pour le calcul d'un module), car les nombres sont des nombres complexes flottants.

Activité 1 (Manipuler les nombres complexes).

Objectifs : faire des calculs avec les nombres complexes.

1. Définis les nombres complexes $z_1 = 1 + 2i$ et $z_2 = 3 - i$. Demande à la machine de calculer :

$$z_1 + z_2 \quad z_1 z_2 \quad z_1^2 \quad |z_1| \quad \frac{1}{z_1}$$

2. Définis le nombre complexe $z = (3 - 4i)^2(2 + i)$. Calcule à la machine la partie réelle de z , sa partie imaginaire et son conjugué.
3. Définis tes propres fonctions pour les opérations sur les nombres complexes. Représente le nombre complexe $z = a + ib$ par le couple de réels (a, b) et $z' = a' + ib'$ par le couple de réels (a', b') (tu n'as pas le droit d'utiliser les nombres complexes de Python).
 - Programme une fonction `addition(a, b, aa, bb)` qui renvoie le couple de réels correspondant au résultat de $(a + ib) + (a' + ib')$.
 - Programme une fonction `multiplication(a, b, aa, bb)` qui renvoie le couple de réels correspondant au résultat de $(a + ib) \times (a' + ib')$.

- Programme une fonction `conjugue(a,b)` qui renvoie le couple de réels correspondant au conjugué de $a + ib$.
- Programme une fonction `module(a,b)` qui renvoie le module de $a + ib$ (c'est un nombre réel).
- Programme une fonction `inverse(a,b)` qui pour $z = a + ib$ teste d'abord si z n'est pas nul et dans ce cas renvoie le couple de réels associé à l'inverse de z en utilisant une des formules :

$$\frac{1}{z} = \frac{\bar{z}}{|z|^2} \quad \text{ou} \quad \frac{1}{z} = \frac{a - ib}{a^2 + b^2}.$$

- Programme une fonction `puissance(a,b,n)` qui pour $z = a + ib$ et $n \geq 0$, renvoie le couple de réels associé à z^n . (*Indications.* Par définition $z^0 = 1$. On pourra construire une boucle et utiliser une des fonctions précédentes.)

Cours 2 (Rappels Matplotlib).

Voici comment afficher des points de coordonnées (x, y) et un segment à l'aide du module `matplotlib`.

```
import matplotlib.pyplot as plt

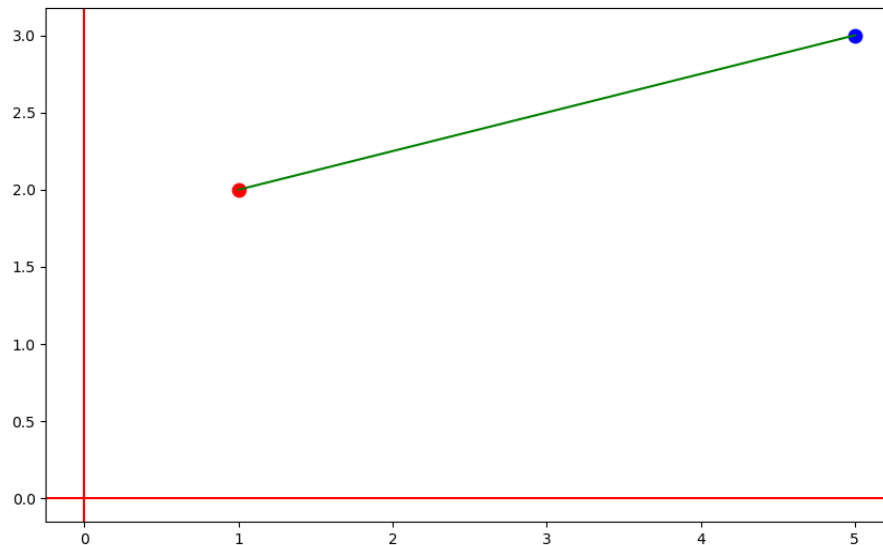
plt.clf() # Efface tout
plt.axhline(y=0, color='r', linestyle='-') # Axe x
plt.axvline(x=0, color='r', linestyle='-') # Axe y
plt.axes().set_aspect('equal') # Repère orthonormé

x1 = 1
y1 = 2
plt.scatter(x1,y1,color='red',s=80) # Un premier point

x2 = 5
y2 = 3
plt.scatter(x2,y2,color='blue',s=80) # Un second point

# Un segment reliant les points
plt.plot([x1,x2],[y1,y2],color='green')

plt.show() # Lancement de la fenêtre
```



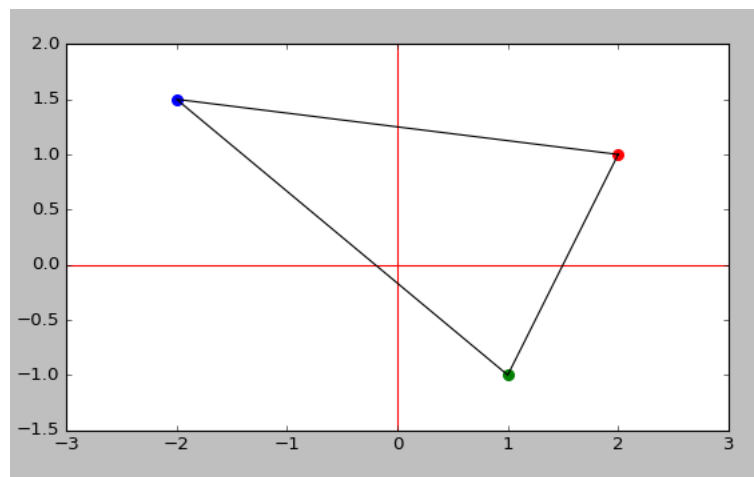
Activité 2 (Visualiser les nombres complexes).

Objectifs : afficher un point connaissant son affixe z .

1. Dessine le point d'affixe 1 et celui d'affixe i . Pour un nombre complexe z , par exemple $z = -2 + 3i$, dessine le point d'affixe z .
2. Pour un nombre complexe z , par exemple $z = 3 - 2i$, dessine les points d'affixes :

$$z \quad 2z \quad iz \quad \bar{z} \quad \frac{z^2}{|z|} \quad \frac{1}{z}$$

3. • Programme une fonction `affiche_triangle(z1, z2, z3)` qui trace le triangle dont les sommets ont pour affixes z_1, z_2, z_3 .



- On fixe $z \in \mathbb{C}$. Quelle semble être la nature du triangle déterminé par $z, 2z, (1 + 2i)z$?
- On pose $\omega = -\frac{1}{2} + \frac{\sqrt{3}}{2}i$. On fixe $z \in \mathbb{C}$. Quelle semble être la nature du triangle donné par $z, \omega z, \omega^2 z$?

Activité 3 (Résolution d'une équation linéaire).

Objectifs : résoudre des équations linéaires en utilisant les nombres complexes. Ici nous allons « hacker » les nombres complexes. En informatique un « hack » est un détournement d'une fonctionnalité.

Résolution. Tu vas programmer une fonction `solution_equation_lineaire(equation)` qui calcule et renvoie la solution d'une équation linéaire. L'équation est donnée en paramètre sous la forme d'une chaîne de caractères. Par exemple avec `"7*x+3 = 0"`, la fonction renvoie `-0.42857...` comme valeur approchée de $-\frac{3}{7}$. Autre exemple avec : `"3*(x+1) + x = 2*x+1"`, la fonction renvoie `-1.0`. Attention : il faut explicitement écrire les multiplications avec le caractère « * ».

Astuce. L'idée est la suivante, une équation linéaire se ramène à la forme :

$$a + bx = 0$$

dont une solution est $-\frac{a}{b}$. L'astuce est de remplacer chaque « x » de l'équation par le nombre complexe i . On obtient ainsi un nombre complexe $a + ib$. En extrayant la partie réelle et la partie imaginaire, on renvoie la solution (réelle) $-a/b$.

Exemple simple. Partant de l'équation $7x + 3 = 0$, on ne garde que la partie gauche de l'équation $7x + 3$. On remplace la lettre x par le nombre complexe i , on obtient $7i + 3$. On extrait sa partie réelle $a = 3$ et sa partie imaginaire $b = 7$. On renvoie la solution $-a/b = -\frac{3}{7}$.

Exemple compliqué. Soit l'équation $3(x + 1) + x = 2x + 1$. Il faut d'abord tout basculer à gauche du signe égal afin de se ramener à l'équation $3(x + 1) + x - (2x + 1) = 0$. On ne garde que l'expression à gauche du signe égal : $3(x + 1) + x - (2x + 1)$. On remplace ensuite la variable x par le nombre complexe i . L'expression devient un nombre complexe $3(i + 1) + i - (2i + 1)$. On note z ce nombre complexe, on calcule sa partie réelle $a = 2$ et sa partie imaginaire $b = 2$. On calcule $-a/b = -1$. La solution de notre équation est donc $x = -1$.

Algorithme.

- — Entrée : une chaîne de caractères représentant une équation linéaire en x .
- — Sortie : la valeur numérique de la solution x .
- Soient G et D les deux chaînes de part et d'autre du signe « = ». (Utilise `equation.split("=")`.)
- Former la chaîne correspondant à la partie gauche moins la partie droite : $G + " - (" + D + ")"$.
- Pour remplacer x par i , il faut remplacer le caractère " x " par la chaîne " $1j$ ". (Utilise `chaîne.replace(mot,nouv_mot)`.) On obtient ainsi une chaîne `z_str`.
- Transformer la chaîne en un nombre complexe par l'opération :

$$z = \text{eval}(z_str)$$
- Calculer la partie réelle a et la partie imaginaire b de z .
- Renvoyer $-a/b$.

Cours 3 (Équation du second degré).

Soit $a, b, c \in \mathbb{R}$ avec $a \neq 0$. On considère l'équation

$$az^2 + bz + c = 0.$$

On note $\Delta = b^2 - 4ac$. Selon le signe de Δ les solutions sont les suivantes :

- $\Delta > 0$: deux solutions réelles $z_1 = \frac{-b + \sqrt{\Delta}}{2a}$ et $z_2 = \frac{-b - \sqrt{\Delta}}{2a}$.
- $\Delta = 0$: solution double $z_0 = \frac{-b}{2a}$.
- $\Delta < 0$: deux solutions complexes $z_1 = \frac{-b + i\sqrt{|\Delta|}}{2a}$ et $z_2 = \frac{-b - i\sqrt{|\Delta|}}{2a}$.

Activité 4 (Équation du second degré).

Objectifs : résoudre les équations du second degré, y compris lorsque le discriminant est négatif.

On considère l'équation :

$$az^2 + bz + c = 0 \quad \text{avec} \quad a, b, c \in \mathbb{R} \quad \text{et} \quad a \neq 0.$$

1. **Équation du second degré.** Programme une fonction `solution_trinome(a,b,c)` qui renvoie les solutions de l'équation sous la forme d'une liste de deux nombres $[z_1, z_2]$ (si la solution est double, renvoie la liste $[z_0, z_0]$).

Exemple. Calcule les solutions de :

$$z^2 - 2z + 1 = 0 \quad z^2 + z - 1 = 0 \quad z^2 + z + 1 = 0$$

2. **Somme et produit.** Lorsque l'on connaît la somme S et le produit P de deux nombres z_1 et z_2 , on peut retrouver z_1 et z_2 . Comment faire ? Réponse : z_1 et z_2 sont les solutions de l'équation :

$$z^2 - Sz + P = 0.$$

Programme une fonction `solution_somme_produit(S,P)` qui renvoie la liste $[z_1, z_2]$ des solutions.

Exemple. Trouve deux nombres dont la somme est 10 et le produit est 20.

3. **Équation bicarrée.** Une équation bicarrée est de la forme :

$$ax^4 + bx^2 + c = 0.$$

Nous étudions seulement les cas où $\Delta = b^2 - 4ac \geq 0$, pour lesquels l'équation admet 4 solutions (réelles ou complexes).

On commence par poser $X = x^2$ et résoudre l'équation du second degré :

$$aX^2 + bX + c = 0.$$

Cette dernière équation admet deux solutions réelles X_1 et X_2 . Pour chacune de ces solutions X :

- si $X \geq 0$, on obtient deux solutions, $+\sqrt{X}$ et $-\sqrt{X}$;
- si $X < 0$, on obtient deux solutions, $+i\sqrt{-X}$ et $-i\sqrt{-X}$.

On obtient ainsi 4 solutions (2 associées à X_1 et 2 associées à X_2).

Programme une fonction `solution_bicarre(a,b,c)` qui renvoie les 4 solutions de l'équation $ax^4 + bx^2 + c = 0$ après avoir vérifié que $\Delta = b^2 - 4ac \geq 0$.

Exemple. Trouve les 4 solutions de l'équation $x^4 - 2x^2 - 3 = 0$.

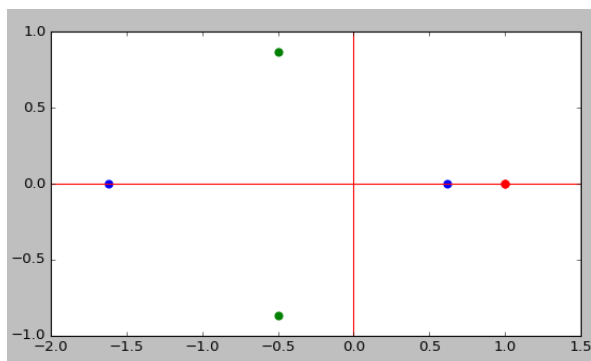
Activité 5 (Famille de racines).

Objectifs : afficher les solutions d'une famille d'équations du second degré.

1. **Afficher les racines.** Programme une fonction `affiche_racines(a,b,c)` qui affiche les deux points correspondant aux deux solutions de l'équation $ax^2 + bx + c = 0$.

Amélioration. C'est mieux d'autoriser en argument optionnel le choix de la couleur du point par une entête `affiche_racines(a,b,c,couleur='red')`.

Retrouve sur la figure ci-dessous les racines des polynômes $x^2 - 2x + 1 = 0$, $x^2 + x - 1 = 0$ et $x^2 + x + 1 = 0$.



2. **Famille de racines.** On considère deux polynômes

$$P_0(x) = x^2 + b_0x + c_0 \quad \text{avec} \quad \Delta_0 = b_0^2 - 4c_0 \leq 0,$$

$$P_1(x) = x^2 + b_1x + c_1 \quad \text{avec} \quad \Delta_1 = b_1^2 - 4c_1 \leq 0.$$

On définit pour $0 \leq t \leq 1$:

$$P_t(x) = (1-t)P_0(x) + tP_1(x) = x^2 + ((1-t)b_0 + tb_1)x + (1-t)c_0 + tc_1.$$

Question. Quelle forme a l'ensemble des racines de la famille $\{P_t(x)\}_{0 \leq t \leq 1}$?

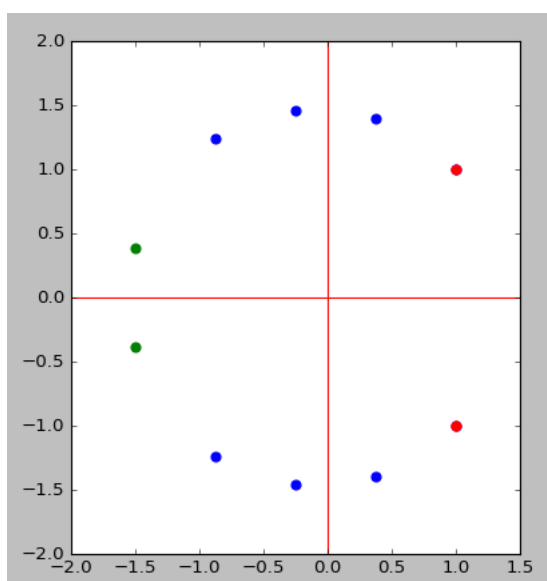
Programme une fonction `affiche_famille(b0,c0,b1,c1)` :

- qui affiche les solutions de $P_0(x) = 0$ (en rouge par exemple),
- qui affiche les solutions de $P_1(x) = 0$ (en vert par exemple),
- qui affiche les solutions de $P_t(x) = 0$ pour n valeurs t , avec $0 \leq t \leq 1$ (en bleu par exemple).

Pour les valeurs de t , tu peux les choisir de la forme k/n avec $0 \leq k < n$.

Amélioration. C'est mieux d'autoriser que n soit un argument optionnel par une entête du type `affiche_famille(b0,c0,b1,c1,n=100)`.

Exemple ci-dessous $P_0(x) = x^2 - 2x + 2$ et $P_1(x) = x^2 + 3x + \frac{12}{5}$, avec seulement $n = 3$ points bleus intermédiaires. Pour répondre à la question il faut afficher plus de points intermédiaires.



Nombres complexes II

On poursuit l'exploration des nombres complexes en se concentrant sur la forme module/argument.

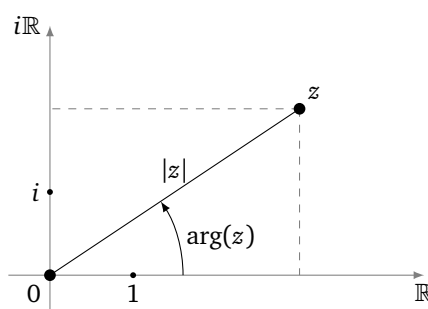
Cours 1 (Nombres complexes).

Module/argument. Tout nombre complexe $z \in \mathbb{C}^*$, s'écrit :

$$z = r(\cos \theta + i \sin \theta)$$

où

- $r = |z|$ est le module de z ,
- et $\theta \in \mathbb{R}$ est un **argument**.



Unicité. Si θ est un argument, alors n'importe quel $\theta + 2k\pi$ est aussi un argument.

Pour éviter cette indétermination, on peut imposer à θ d'appartenir à l'intervalle $]-\pi, +\pi]$, l'argument est alors unique. Pour $z \in \mathbb{C}^*$, il existe un unique couple (r, θ) avec $r > 0$ et $\theta \in]-\pi, +\pi]$ tel que :

$$z = r(\cos \theta + i \sin \theta).$$

Remarques.

- Une autre convention aurait été de choisir l'intervalle $[0, 2\pi[$.
- L'écriture (r, θ) s'appelle aussi l'écriture en coordonnées polaires d'un nombre complexe, par opposition à l'écriture $z = a + ib$ qui est l'écriture cartésienne.

Cours 2 (Module cmath).

Le module `cmath` fournit des outils supplémentaires pour les nombres complexes. Pour éviter les conflits avec le module `math` nous l'importerons par :

```
import cmath
```

1. `cmath.phase(z)` renvoie l'argument $\theta \in]-\pi, +\pi]$ du nombre complexe z . Exemple : `cmath.phase(1-1j)` renvoie `-0.785...` qui correspond à la valeur $-\frac{\pi}{4}$.
2. Rappel : `abs(z)` renvoie le module $|z|$ (c'est une fonction interne à Python).

3. `cmath.polar(z)` renvoie le couple module/argument (r, θ) . Exemple : `cmath.polar(1-1j)` renvoie $(1.414\dots, -0.785\dots)$ qui correspond au couple $(r, \theta) = (\sqrt{2}, -\frac{\pi}{4})$.
4. `cmath.rect(r, theta)` renvoie le nombre complexe dont le module est r et l'argument θ . Exemple : `cmath.rect(2, pi/4)` renvoie $1.414\dots + 1.414\dots j$ et correspond à $\sqrt{2} + i\sqrt{2}$.

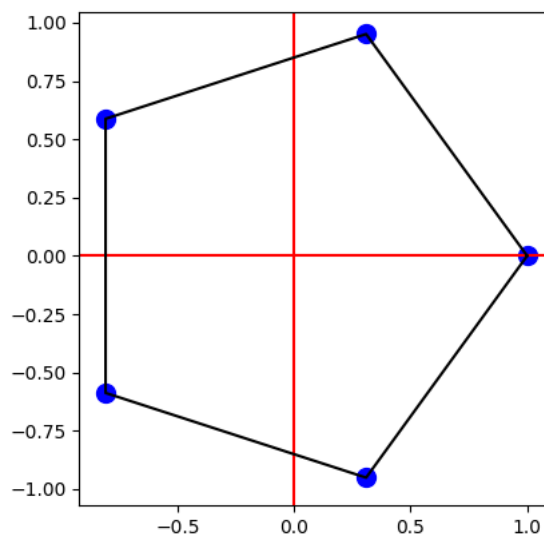
Activité 1 (Module/argument).

Objectifs : utiliser Python pour calculer et mieux comprendre la forme module/argument.

1. Pour un nombre complexe z , par exemple $z = 1 + 3i$ ou $z = 1 + i$, calcule son module et son argument à l'aide de Python.
2. Quel nombre complexe a pour module 2 et argument $\frac{\pi}{3}$? Même question pour le complexe de module 3 et d'argument $\frac{3\pi}{2}$. Essaie de deviner la réponse exacte à partir des valeurs approchées données par Python.
3. À l'aide du module `matplotlib`, place le point d'affixe z dont on te donne le module et l'argument, par exemple de module $\sqrt{2}$ et argument $\frac{\pi}{6}$.
4. Soit $n \geq 3$. Soit ω le nombre complexe de module 1 et d'argument $\frac{2\pi}{n}$. Trace le polygone ayant pour sommets les points d'affixes :

$$1, \omega, \omega^2, \dots, \omega^{n-1}.$$

Quelle est la nature de ce polygone ?



Activité 2 (Module/argument (suite)).

Objectifs : créer tes propres fonctions qui permettent la conversion entre l'écriture cartésienne d'un nombre complexe et son écriture sous la forme module/argument.

Tu vas écrire tes propres fonctions pour calculer avec les modules et les arguments.

1. Programme une fonction `polaire_vers_cartesien(module, argument)` qui renvoie le nombre complexe z (sous la forme d'un nombre complexe Python) dont le module et l'argument sont donnés. Utilise la formule

$$z = r \cos \theta + ir \sin \theta.$$

Compare ton résultat avec la fonction `rect` du module `cmath`.

2. Programme une fonction `cartesien_vers_polaire(z)` qui renvoie le module et l'argument du nombre complexe z . Récupère d'abord la partie réelle x et la partie imaginaire y de z . Le module est alors facile à calculer. L'argument se calcule par la formule :

$$\theta = \text{atan2}(y, x)$$

La fonction `atan2` est une variante de la fonction « arctangente » et est disponible dans le module `math`.

Compare ta fonction avec les fonctions `phase` et `polar` du module `cmath`.

3. Programme une fonction `argument_dans_intervalle(angle)` qui renvoie une mesure de l'angle dans l'intervalle $]-\pi, +\pi]$. Par exemple soit $\theta = \frac{5\pi}{2}$, comme $\theta = -\frac{\pi}{2} + 3 \cdot 2\pi$ alors $\theta' = -\frac{\pi}{2}$ est la mesure de l'angle dans l'intervalle $]-\pi, +\pi]$.

Indication. Commence par ramener l'angle dans l'intervalle $[0, 2\pi[$, puis discute selon la valeur.

Une fois terminé compare ton résultat avec la commande `angle % 2*pi`.

Cours 3 (Notation exponentielle).

- **Notation exponentielle.** On note

$$e^{i\theta} = \cos(\theta) + i \sin(\theta).$$

C'est donc le nombre complexe de module 1 et d'argument θ .

- **Formules d'Euler.** Un petit calcul conduit à :

$$\cos \theta = \frac{e^{i\theta} + e^{-i\theta}}{2} \quad \text{et} \quad \sin \theta = \frac{e^{i\theta} - e^{-i\theta}}{2i}.$$

- **Formule de Moivre.**

$$(\cos \theta + i \sin \theta)^n = \cos(n\theta) + i \sin(n\theta).$$

Avec la notation exponentielle, l'écriture de cette formule est très simple :

$$(e^{i\theta})^n = e^{in\theta}.$$

Activité 3 (Euler, de Moivre, Gauss).

Objectifs : mettre en œuvre plusieurs formules.

1. **Euler.** Programme deux fonctions `cosinus(t)` et `sinus(t)` qui calculent et renvoient le cosinus et le sinus d'un réel t donné en utilisant les formules d'Euler.

Indication. Utilise ta fonction `polaire_vers_cartesien()` de l'activité 1 pour calculer e^{it} .

Exemple. Retrouve le sinus et le cosinus de $t = \frac{\pi}{6}$.

2. **de Moivre.** Programme une fonction `puissance_bis(z,n)` qui calcule z^n à l'aide de la formule de Moivre selon le principe suivant :

- Écrire z sous la forme $z = re^{i\theta}$ (utilise ta fonction `cartesien_vers_polaire()`).
- Calculer r^n et $n\theta$.
- Renvoyer z^n grâce à la formule de Moivre $z^n = r^n e^{in\theta}$ (utilise ta fonction `polaire_vers_cartesien()`).

Exemple. Calcule $(2 - 3i)^{10}$.

Complexité. La formule de Moivre permet de remplacer le calcul d'une puissance d'un nombre complexe par le calcul de la puissance d'un nombre réel (son module).

3. **Gauss.** Comment calculer plus rapidement le produit de deux nombres complexes ? Soit $z = a + ib$

et $z' = c + id$. La formule naïve donnée par la définition est :

$$z \times z' = (ac - bd) + i(ad + bc).$$

Pour calculer un produit de deux nombres complexes, il faut donc calculer le produit de 4 nombres réels : ac , bd , ad , bc .

Nous allons voir deux méthodes, dues à Gauss, qui ne nécessitent que 3 multiplications de nombres réels.

Méthode 1. Calculer $r = ac$, $s = bd$, $t = (a + b)(c + d)$, alors $z = (r - s) + i(t - r - s)$.

Méthode 2. Calculer $r = c(a + b)$, $s = a(d - c)$, $t = b(c + d)$, alors $z = (r - t) + i(r + s)$.

Programme trois fonctions du type `multiplication(a,b,c,d)` qui renvoient la partie réelle et la partie imaginaire de $(a + ib) \times (c + id)$ par les trois méthodes décrites ici. Teste tes fonctions en calculant $(2 + 5i) \times (3 - 2i)$.

Activité 4 (Cercles et droites).

Objectifs : tracer des cercles et des droites en utilisant les nombres complexes.

1. Programme une fonction `affiche_liste(zliste)` qui trace et affiche les points d'affixe z donnés dans la liste.

Indication. Utilise la commande `plt.scatter(x,y)` provenant de `matplotlib`. C'est encore mieux si tu autorises les arguments optionnels avec une entête du type `affiche_liste(zliste, couleur='blue', taille=10)`.

2. Programme une fonction `trace_cercle(z0,r)` qui renvoie une liste de complexes z appartenant au cercle centré en z_0 et de rayon r .

Indications.

- Ces complexes z vérifient $|z - z_0| = r$ et sont donc de la forme :

$$z = re^{2i\pi\theta}, \quad 0 \leq \theta < 1.$$

- C'est mieux d'avoir en argument optionnel le nombre de points avec une entête du type `trace_cercle(z0,r,numpoints=100)`.
- Trace le cercle à l'aide de ta fonction `affiche_liste()`.

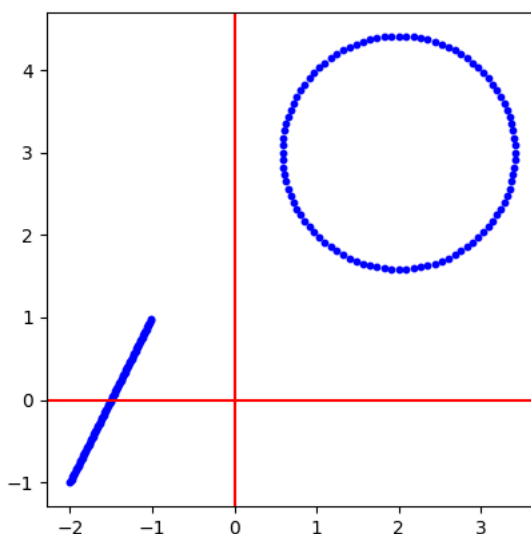


Figure. Voici le cercle de centre $2 + 3i$ et de rayon $\sqrt{2}$, ainsi que le segment entre les points d'affixes $-2 - i$ et $-1 + 3i$.

3. Programme une fonction `trace_segment(z0, z1)` qui renvoie une liste de complexes z appartenant au segment $[z_0, z_1]$.

Indications.

- Ces complexes z vérifient $z \in [z_0, z_1]$ et sont donc de la forme :

$$z = (1 - t)z_0 + tz_1, \quad 0 \leq t \leq 1.$$

- C'est mieux d'avoir le nombre de points en argument optionnel avec une entête du type `trace_segment(z0, z1, numpoints=100)`.
- Trace le segment à l'aide de ta fonction `affiche_liste()`.

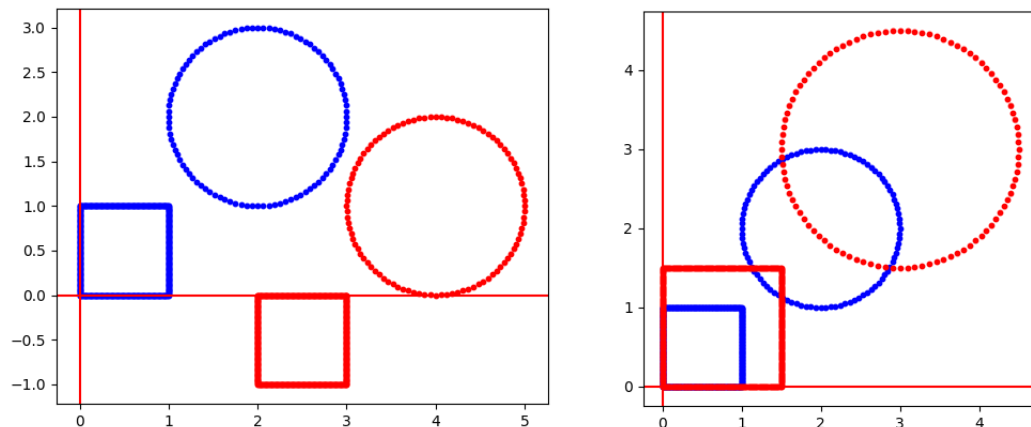
Activité 5 (Transformations du plan).

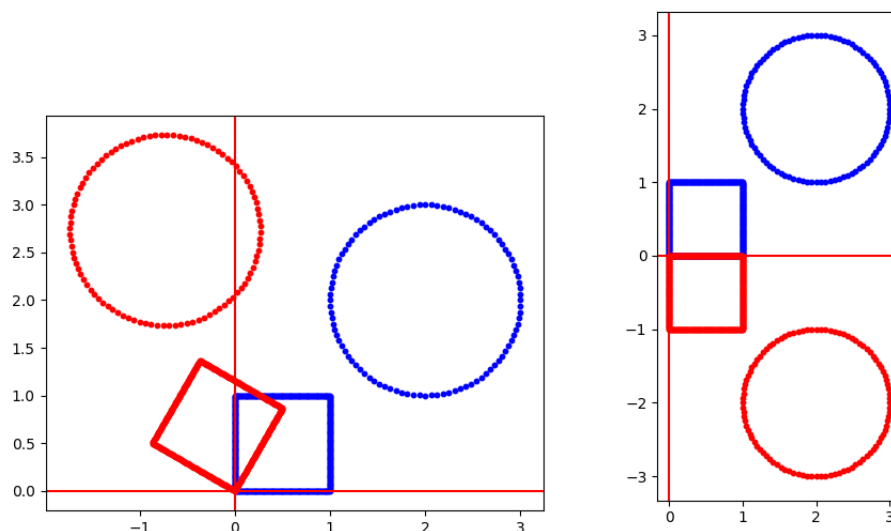
Objectifs : définir des transformations du plan à l'aide des nombres complexes.

1. Programme les fonctions suivantes. Chaque fonction est du type `transformation(zliste)` et renvoie la liste des $f(z)$ pour z parcourant la liste donnée :

- une fonction `translation(zliste, v)` qui correspond à la translation $z \mapsto z + v$, où $v \in \mathbb{C}$ est fixé,
- une fonction `homothetie(zliste, k)` qui correspond à l'homothétie de centre 0 et de rapport $k \in \mathbb{R} : z \mapsto kz$,
- une fonction `rotation(zliste, theta)` qui correspond à la rotation d'angle θ , centrée en 0 : $z \mapsto ze^{i\theta}$,
- une fonction `symetrie(zliste)` qui correspond à la symétrie axiale $z \mapsto \bar{z}$.

Affiche ensuite l'image d'un cercle et d'un carré pour chacune de ces transformations (un carré est formé de quatre segments!). Ci-dessous un cercle et un carré (en bleu) et leur image pour chaque transformation (en rouge).

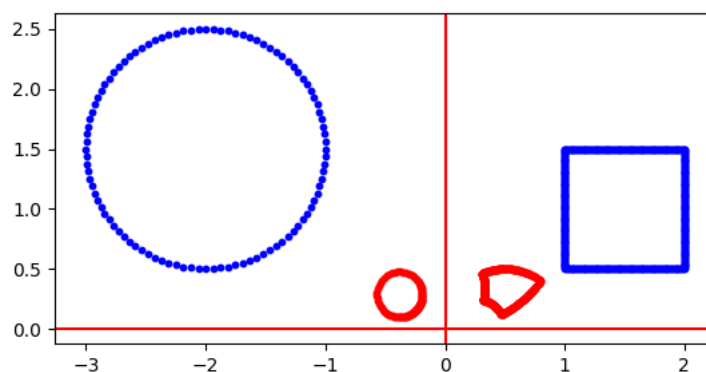




2. Programme une fonction `inversion(zliste)` qui correspond à l'inversion qui est l'application $z \mapsto \frac{1}{z}$ (pour $z \in \mathbb{C}^*$).

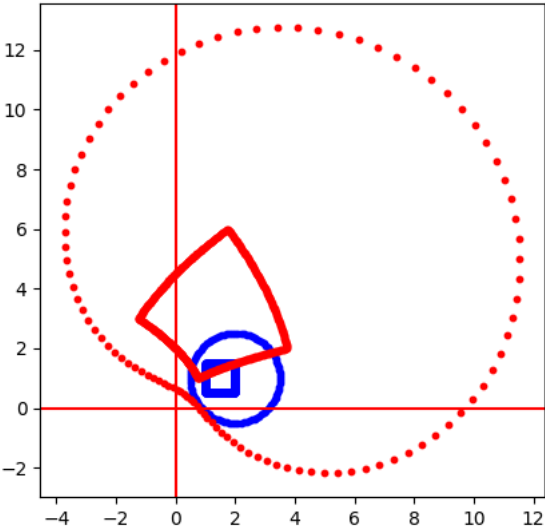
En particulier essaie de conjecturer en quoi est transformée une droite, en quoi est transformé un cercle (les cas où la droite ou le cercle passent par l'origine sont spéciaux).

Ci-dessous un cercle et un carré (en bleu) et leur image par l'inversion (en rouge).



3. Programme une fonction `au_carre(zliste)` qui correspond à l'application $z \mapsto z^2$.

Ci-dessous un cercle et un carré (en bleu) et leur image (en rouge).



Dérivée – Zéros de fonctions

Nous étudions les fonctions : le calcul de la dérivée d'une fonction, le tracé du graphe et de tangentes, et enfin la recherche des valeurs en lesquelles la fonction s'annule.

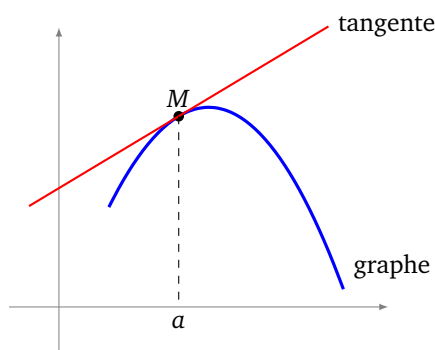
Cours 1 (Dérivée).

Par définition le nombre dérivé de f en a (s'il existe) est :

$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}.$$

Dans cette fiche nous supposons que toutes les dérivées étudiées existent.

Voici l'interprétation géométrique du nombre dérivé : $f'(a)$ est le coefficient directeur de la tangente au graphe de f au point d'abscisse a .



Cours 2 (Fonction lambda).

Une fonction lambda (lettre grecque λ) est une façon simple de définir une fonction en Python qui s'apparente à une fonction mathématique. Par exemple :

```
f = lambda x: x**2
```

Cela définit une fonction Python f qui correspond à la fonction mathématique f définie par $f : x \mapsto x^2$.

Ainsi $f(2)$ renvoie 4, $f(3)$ renvoie 9...

C'est une alternative condensée au code suivant :

```
def f(x):  
    return x**2
```

Une fonction est un objet Python comme un autre. Elle peut donc être utilisée dans le programme comme dans l'exemple suivant qui teste si $f(a) > f(b)$:


```
def est_plus_grand(f,a,b):
    if f(a) > f(b):
        return True
    else:
        return False
```

Pour les deux fonctions f définies au-dessus (soit à l'aide de `lambda`, soit à l'aide de `def`) alors
`est_plus_grand(f, 1, 2)`

renvoie « Faux ».

À l'aide des fonctions `lambda` on peut aussi se permettre de ne pas donner de nom à une fonction, comme ci-dessous avec la fonction $x \mapsto \frac{1}{x}$. Alors

```
est_plus_grand(lambda x: 1/x, 1, 2)
```

qui renvoie « Vrai » (`lambda x: 1/x` joue le rôle de f).

Activité 1 (Calcul de la dérivée en un point).

Objectifs : calculer une valeur approchée de la dérivée en un point.

On va calculer une valeur approchée du nombre dérivé $f'(a)$ en calculant le taux d'accroissement $\frac{f(a+h)-f(a)}{h}$ avec h suffisamment petit.

1. Définis la fonction $f, x \mapsto x\sqrt{1-x}$. Deux méthodes : soit à l'aide de `def f(x): ...`, soit par `f = lambda x: ...`. Calcule les valeurs approchées de $f(k)$ pour $k \in \{0, 1, 2, \dots, 5\}$.
2. Programme une fonction `derivee(f, a)` qui calcule une valeur approchée de la dérivée de f en a par la formule

$$f'(a) \simeq \frac{f(a+h)-f(a)}{h}$$

en prenant par exemple pour valeur $h = 0.0001$.

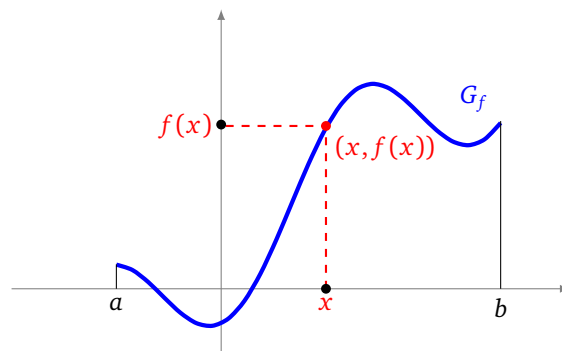
Pour la fonction $f : x \mapsto x^3$, compare la valeur approchée en a que tu obtiens avec la valeur exacte de $f'(k)$ pour $k \in \{0, 1, 2, \dots, 5\}$. Diminue la valeur de h pour obtenir une meilleure approximation.

Activité 2 (Graphe d'une fonction et tangente).

Objectifs : tracer le graphe d'une fonction ainsi que des tangentes.

Soit $f : [a, b] \rightarrow \mathbb{R}$ une fonction. Son graphe G_f est :

$$G_f = \{(x, f(x)) \mid x \in [a, b]\}$$

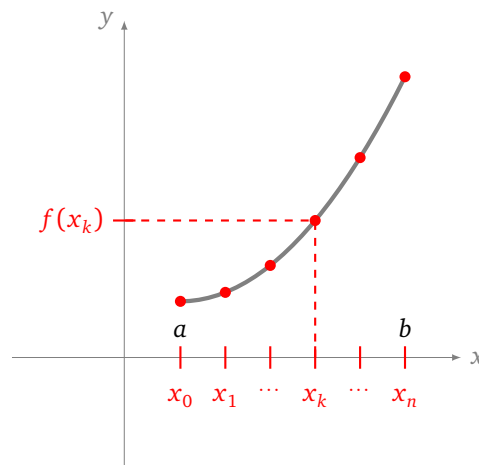


1. **Calculer des points.** Soit f une fonction définie sur un intervalle $[a, b]$. On divise l'intervalle $[a, b]$ en n sous-intervalles de longueur $\frac{b-a}{n}$ en définissant :

$$x_k = a + k \frac{b-a}{n}.$$



Programme une fonction `graphe(f, a, b, n)` qui calcule et renvoie la liste des points $(x_k, f(x_k))$ pour $k = 0, \dots, n$.

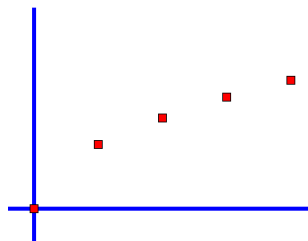


Par exemple pour $f = \text{lambda } x: x*x$ alors `graphe(f, 0, 2, 4)` renvoie la liste :

$[(0, 0), (0.5, 0.25), (1.0, 1.0), (1.5, 2.25), (2.0, 4.0)]$

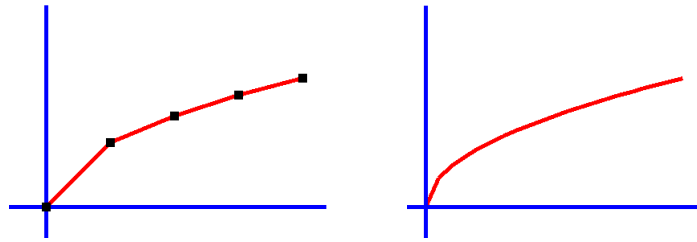
2. **Afficher des points.** Programme une fonction `afficher_points(points)` qui affiche une liste de points.

Indications. Tu peux utiliser le module `tkinter` (ou bien le module `matplotlib`). Tu peux utiliser une variable `echelle` pour contrôler la taille de l'affichage. Les figures ci-dessous sont tracées pour la fonction f définie par $f(x) = \sqrt{x}$ sur l'intervalle $[0, 4]$.



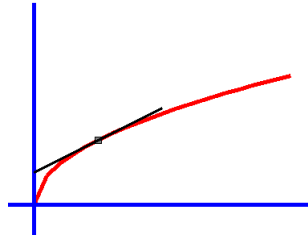
3. **Tracer le graphe.** Améliore la fonction précédente pour écrire une fonction `tracer_graphe(f, a, b)` qui trace le graphe de f .

Indications. Il suffit de relier n points du graphe entre eux pour n assez grand (avec 5 points à gauche et 20 points à droite).



4. Tracer une tangente.

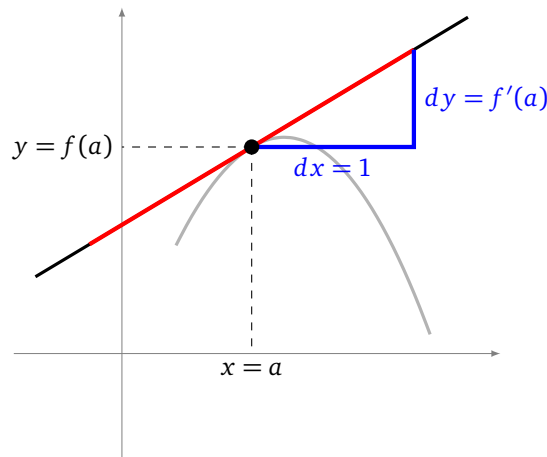
Trace la tangente au graphe au point $(a, f(a))$ par une fonction `tracer_tangente(f, a)`.



Indications. On se place au point $(x, y) = (a, f(a))$. En ce point la pente de la tangente est donnée par $f'(a)$. En posant

$$dx = 1 \quad \text{et} \quad dy = f'(a)$$

alors on représente une demi-tangente par le segment reliant (x, y) à $(x + dx, y + dy)$. L'autre demi-tangente est représentée par le segment reliant (x, y) à $(x - dx, y - dy)$.



Cette activité peut être l'occasion d'utiliser les arguments optionnels, par exemple au lieu de définir la fonction `tracer_graphe()` par l'entête :

```
def tracer_graphe(f, a, b):
```

et d'avoir des variables locales `n` et `echelle`, tu peux définir ta fonction par :

```
def tracer_graphe(f, a, b, n=20, echelle=50):
```

Ce qui permet d'avoir une valeur de `n` et de `echelle` par défaut, en conservant la possibilité de les changer. Des appels possibles sont :

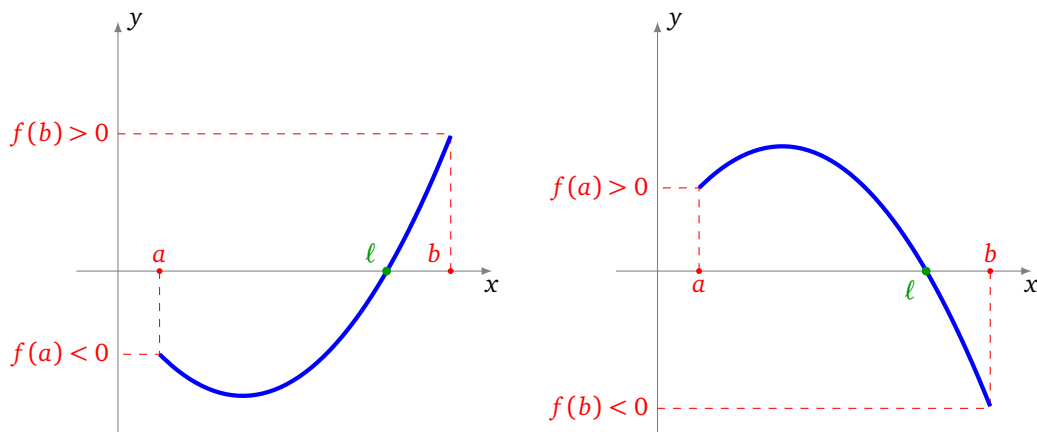
- `tracer_graphe(f, a, b);`
- `tracer_graphe(f, a, b, n=100)` pour tracer plus de points;
- `tracer_graphe(f, a, b, echelle=10)` pour changer l'échelle.



Cours 3 (Dichotomie).

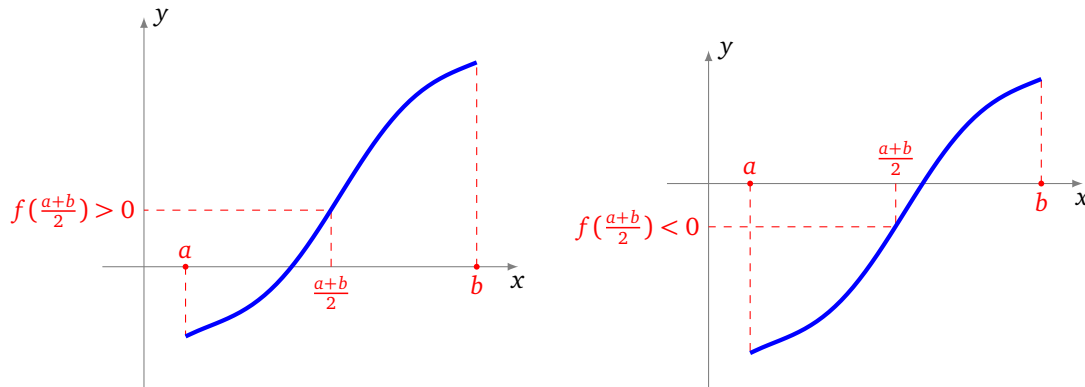
Le méthode de dichotomie est basée sur cette version du théorème des valeurs intermédiaires.

Théorème. Soit $f : [a, b] \rightarrow \mathbb{R}$ une fonction continue. Si $f(a)$ et $f(b)$ sont de signes contraires, alors f s'annule au moins une fois sur l'intervalle $[a, b]$. Autrement dit, il existe $\ell \in [a, b]$ tel que $f(\ell) = 0$.



Principe de la dichotomie (διχοτομία signifie « coupé en deux »). On sait que notre fonction f s'annule sur $[a, b]$. On calcule $f\left(\frac{a+b}{2}\right)$, c'est-à-dire l'image du milieu du segment $[a, b]$. On cherche ensuite où f peut s'annuler par rapport à ce milieu :

- Si $f(a)$ et $f\left(\frac{a+b}{2}\right)$ sont de signes contraires alors f s'annule sur $\left[a, \frac{a+b}{2}\right]$,
- sinon f s'annule sur $\left[\frac{a+b}{2}, b\right]$.



On recommence l'opération sur l'intervalle $\left[a, \frac{a+b}{2}\right]$ ou bien sur l'intervalle $\left[\frac{a+b}{2}, b\right]$.

Remarques :

- $f(a)$ et $f(b)$ sont de signes contraires si et seulement si $f(a) \times f(b) \leq 0$.
- On va construire des intervalles de plus en plus petits qui contiennent une solution ℓ , de $f(\ell) = 0$. On obtient donc un encadrement de ℓ (mais pas sa valeur exacte).
- Il se peut que f s'annule plusieurs fois, mais la méthode de dichotomie ne fournit l'encadrement que d'une seule solution.
- Pour expliquer la partie « si » du principe de la dichotomie, on applique le théorème des valeurs intermédiaires sur $\left[a, \frac{a+b}{2}\right]$. Pour expliquer la partie « sinon », on remarque d'abord que $f(b)$ et $f\left(\frac{a+b}{2}\right)$ doivent être de signes contraires, puis on applique le théorème des valeurs intermédiaires.

Exemple.

On cherche « à la main » une valeur approchée de $\sqrt{2}$.

- Soit f définie par $f(x) = x^2 - 2$. On se place sur l'intervalle $[1, 2]$.
- Comme $f(1) = -1 \leq 0$ et $f(2) = 2 \geq 0$ et que f est continue alors f s'annule sur l'intervalle $[1, 2]$ par le théorème des valeurs intermédiaires. Bien sûr, ici f s'annule en $\ell = \sqrt{2}$. Pour l'instant on a prouvé : $1 \leq \sqrt{2} \leq 2$.
- On divise l'intervalle $[1, 2]$ en deux parties, le milieu étant $\frac{3}{2}$, on calcule :

$$f\left(\frac{3}{2}\right) = \left(\frac{3}{2}\right)^2 - 2 = \frac{1}{4} \geq 0.$$

Donc sur le demi-intervalle $[1, \frac{3}{2}]$ on a $f(1) \leq 0$ et $f(\frac{3}{2}) \geq 0$, et c'est bien là que f s'annule. Autrement dit $1 \leq \sqrt{2} \leq \frac{3}{2}$. C'est un encadrement deux fois plus précis qu'auparavant.

- On divise l'intervalle $[1, \frac{3}{2}]$ en deux parties, le milieu étant $\frac{5}{4}$, on calcule :

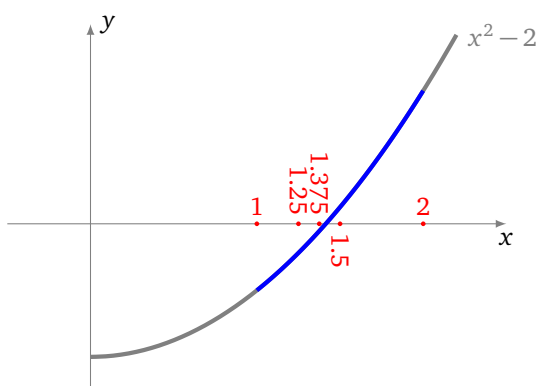
$$f\left(\frac{5}{4}\right) = \left(\frac{5}{4}\right)^2 - 2 = -\frac{7}{16} \leq 0.$$

Donc sur le demi-intervalle $[\frac{5}{4}, \frac{3}{2}]$ on a $f(\frac{5}{4}) \leq 0$ et $f(\frac{3}{2}) \geq 0$, ainsi $\frac{5}{4} = 1.25 \leq \sqrt{2} \leq \frac{3}{2} = 1.5$.

- On continue ainsi, on obtient des intervalles $[a_i, b_i]$ de plus en plus petits qui encadrent $\sqrt{2}$:

$a_0 = 1$	$b_0 = 2$
$a_1 = 1$	$b_1 = 1.5$
$a_2 = 1.25$	$b_2 = 1.5$
$a_3 = 1.375$	$b_3 = 1.5$
$a_4 = 1.375$	$b_4 = 1.4375$
$a_5 = 1.40625$	$b_5 = 1.4375$
$a_6 = 1.40625$	$b_6 = 1.421875$
$a_7 = 1.4140625$	$b_7 = 1.421875$
$a_8 = 1.4140625$	$b_8 = 1.41796875$

Donc en 8 étapes on prouve que $1.4140625 \leq \sqrt{2} \leq 1.41796875$. En particulier on obtient les deux premières décimales de $\sqrt{2}$: $\sqrt{2} = 1.41 \dots$

**Activité 3 (Dichotomie).**

Objectifs : trouver une solution approchée d'une équation $f(x) = 0$.

Le principe de la dichotomie se décline en l'algorithme suivant :

Algorithme.

- — Entrée : une fonction f , un intervalle $[a, b]$ avec $f(a) \cdot f(b) \leq 0$, une marge d'erreur ϵ .
- Sortie : un intervalle $[a', b']$ tel que $|b' - a'| \leq \epsilon$ sur lequel f s'annule, autrement dit, il existe $a' \leq \ell \leq b'$ tel que $f(\ell) = 0$.
- Tant que $|b - a| > \epsilon$:
 - poser $c = \frac{a+b}{2}$,
 - si $f(a) \times f(c) \leq 0$, faire $b \leftarrow c$,
 - sinon, faire $a \leftarrow c$.
- À la fin renvoyer a et b (qui encadrent la solution).

1. Programme cet algorithme en une fonction `dichotomie(f, a, b, epsilon)`.

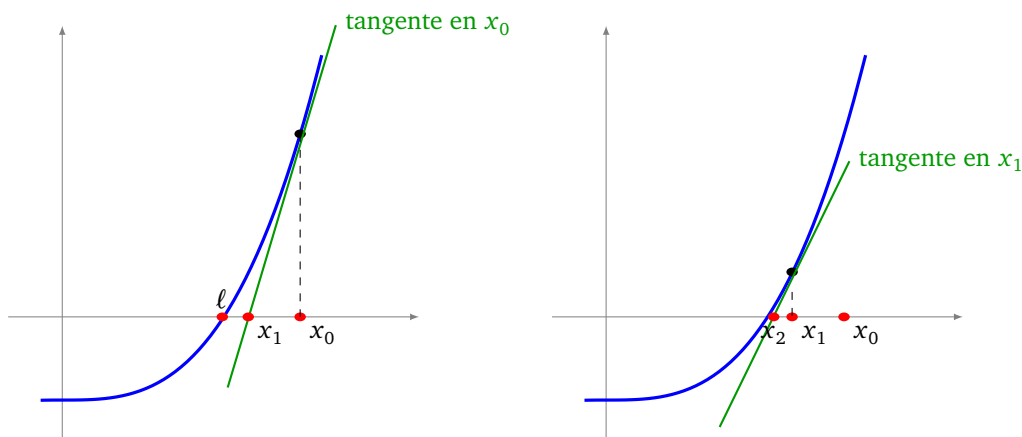
2. Exemples.

- (a) Trouve une valeur approchée de $\sqrt{3}$ à 10^{-3} près, en utilisant la fonction définie par $f(x) = x^2 - 3$ sur l'intervalle $[1, 2]$.
 - (b) Trouve une valeur approchée de $\sqrt[3]{5}$, en utilisant la fonction définie par $f(x) = x^3 - 5$.
 - (c) Trouve une valeur approchée de chacune des trois solutions de l'équation $x^5 - 3x + 1 = 0$.
3. Soit la fonction définie par $f(x) = x^2 - 3$ sur l'intervalle $[1, 2]$. Combien faut-il d'étapes pour obtenir une approximation de $\sqrt{3}$ avec 10 décimales exactes après la virgule ?

Cours 4 (Méthode de Newton).

On va voir une autre méthode très efficace pour obtenir une valeur approchée d'une solution ℓ de $f(\ell) = 0$. L'idée de la méthode de Newton est d'utiliser la tangente :

- on part d'une valeur x_0 quelconque,
- on trace la tangente au graphe de f au point d'abscisse x_0 ,
- cette tangente recoupe l'axe des abscisses en un point d'abscisse x_1 (figure de gauche),
- cette valeur x_1 est plus proche de ℓ que x_0 ,
- on recommence à partir de x_1 : on trace la tangente, elle recoupe l'axe des abscisses, on obtient une valeur $x_2 \dots$ (figure de droite).



On va ainsi définir une suite (x_n) par récurrence. L'équation de la tangente en une valeur x_n est donnée par $y - f(x_n) = f'(x_n)(x - x_n)$. En partant d'une valeur x_0 , on obtient une formule de récurrence, pour

$n \geq 0$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Pour que cette méthode fonctionne il faut tout de même partir d'une valeur x_0 pas trop éloignée de la solution ℓ cherchée.

Exemple.

On cherche encore « à la main » une valeur approchée de $\sqrt{2}$.

- Soit f définie par $f(x) = x^2 - 2$. On a donc $f'(x) = 2x$.
- On part de $x_0 = 2$.
- On calcule $f(x_0) = 2$ et $f'(x_0) = 4$. Par la formule de récurrence :

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = \frac{3}{2} = 1.5$$

- On calcule $f(\frac{3}{2}) = \frac{1}{4}$, $f'(\frac{3}{2}) = 3$ et donc

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = \frac{17}{12} = 1.41666\dots$$

- Puis $x_3 = 1.4142156\dots$ qui a déjà 5 chiffres après la virgule de corrects !

Activité 4 (Méthode de Newton).

Objectifs : programmer la méthode de Newton.

Algorithme.

- Entrée : une fonction f , une valeur de départ a , un nombre d'itérations n .
- Sortie : une valeur approchée de ℓ tel que $f(\ell) = 0$.
- Poser $x = a$.
- Répéter n fois :

$$x \leftarrow x - \frac{f(x)}{f'(x)}$$

- À la fin renvoyer x (qui approche une solution).

1. Programme cet algorithme en une fonction `newton(f, a, n)`.

Indication. Utilise ta fonction `derivee(f, x)` avec un h très petit.

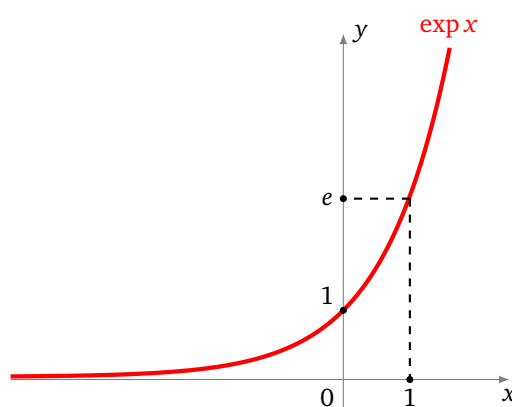
2. *Exemples.*

- (a) Trouve une valeur approchée de $\sqrt{3}$ à 10^{-3} près, en utilisant la fonction définie par $f(x) = x^2 - 3$, en partant de $a = 2$.
- (b) Trouve une valeur approchée de $\sqrt[3]{5}$, en utilisant la fonction définie par $f(x) = x^3 - 5$.
- (c) Trouve une valeur approchée de la solution de l'équation $\cos(x) = x$.

3. Soit la fonction définie par $f(x) = x^2 - 3$ et $a = 2$. Combien faut-il d'étapes pour obtenir une approximation de $\sqrt{3}$ avec 10 décimales exactes après la virgule ? Compare avec la méthode de la dichotomie !

Exponentielle

L'exponentielle joue un rôle important dans la vie de tous les jours : elle permet de modéliser la vitesse de refroidissement de votre café, de calculer la croissance d'une population ou de calculer la performance d'un algorithme.



Cours 1 (La fonction exponentielle).

Voici un très court cours sur l'exponentielle.

- La **fonction exponentielle** est la fonction $\exp : \mathbb{R} \rightarrow]0, +\infty[$ qui vérifie :

$$\exp(0) = 1 \quad \text{et} \quad \exp(x + y) = \exp(x) \times \exp(y).$$
- On note $e = \exp(1) = 2.718281\dots$
- La fonction exponentielle est strictement croissante, strictement positive, $\lim_{x \rightarrow -\infty} \exp(x) = 0$, $\lim_{x \rightarrow +\infty} \exp(x) = +\infty$.
- $\exp(-x) = \frac{1}{\exp(x)}$, $\exp(nx) = (\exp(x))^n$.
- On note $e^x = \exp(x)$, de sorte que $e^{x+y} = e^x \cdot e^y$, $e^{-x} = 1/e^x$, $e^0 = 1$, $e^1 = e$, $e^{nx} = (e^x)^n \dots$
- La dérivée de l'exponentielle est elle-même : $\exp'(x) = \exp(x)$.
- La **fonction logarithme** $\ln :]0, +\infty[\rightarrow \mathbb{R}$ est la bijection réciproque de la fonction exponentielle, c'est-à-dire :

$$y = \exp(x) \iff x = \ln(y).$$

Plus précisément :

$$\begin{aligned} \exp(\ln(x)) & \quad \text{pour tout } x > 0, \\ \ln(\exp(x)) & \quad \text{pour tout } x \in \mathbb{R}. \end{aligned}$$

Le logarithme vérifie $\ln(1) = 0$ et $\ln(x \times y) = \ln(x) + \ln(y)$.

- L'exponentielle permet de définir une puissance avec des exposants réels :

$$a^b = \exp(b \ln(a)).$$

Autrement dit $a^b = e^{b \ln(a)}$.

Cours 2 (Exponentielle et logarithme avec Python).

- Pour obtenir une valeur approchée de l'exponentielle en un point il faut importer le module `math` par la commande `from math import *` puis utiliser la fonction `exp()`.
- Il y a plusieurs fonctions logarithmes accessibles depuis le module `math`. Celle qui correspond au logarithme népérien $\ln(x)$ s'obtient par l'appel à la fonction `log()`. (À ne pas confondre avec la notation mathématique $\log(x)$ qui désigne habituellement le logarithme décimal !)
- Il est possible de faire des calculs de puissances, sans importer le module `math`, par la commande :

$$a ** b$$

Objectifs des quatre premières activités : découvrir le comportement de l'exponentielle à travers des activités variées.

Activité 1 (Les grains de riz).

Pour le remercier d'avoir inventé le jeu d'échec, le roi des Indes demande à Sissa ce qu'il veut comme récompense. Sissa répond : « Je souhaiterais que vous déposiez un grain de riz sur la première case, deux grains de riz sur la seconde, quatre grains de riz sur la troisième... et ainsi de suite en doublant à chaque case le nombre de grains. ». « Facile ! » répondit le roi...

1. Combien faut-il de grains de riz au total pour recouvrir l'échiquier de 64 cases ?
2. Un kilogramme de riz contient 50 000 grains. Quelle est la masse totale (en tonnes) de tous les grains de riz de l'échiquier ?

Et toi : préfères-tu recevoir 1 million d'euros d'un coup ou bien 1 centime le premier jour, puis 2 centimes le second, 4 centimes le jour suivant... pendant un mois ?

Activité 2 (Le nénuphar qui s'agrandit).

Un nénuphar multiplie sa surface d'un facteur 1.5 chaque jour. Au dixième jour sa surface vaut 100 m^2 .

1. Quelle surface recouvre le nénuphar au quinzième jour ?
2. Quelle surface S_9 recouvrait le nénuphar le neuvième jour ? Et le huitième jour ? Calcule la surface S_0 que recouvrait le nénuphar le jour initial (le jour 0).
3. Trouve la formule $S(j)$ qui exprime la surface recouverte au jour j , en fonction de j et de S_0 .
 Définis une fonction `surface_nenuphar(x)` qui renvoie cette surface $S(x)$. Le paramètre x représente le nombre de jours écoulés, mais n'est pas nécessairement un nombre entier.
 Vérifie que tu peux utiliser indifféremment une commande du type `a ** x` (pour a^x) ou bien `exp(x*log(a))` pour $\exp(x \ln(a))$.
4. Trouve par tâtonnement ou par balayage au bout de combien de jours la surface du nénuphar est de $10\,000 \text{ m}^2$. Donne la réponse avec deux chiffres exacts après la virgule.
5. (Si tu maîtrises le logarithme.) Trouve l'expression de x en fonction de la surface couverte S . Programme une fonction `jour_nenuphar(S)` qui renvoie le nombre de jours écoulés x pour atteindre la surface S donnée. Par exemple `jour_nenuphar(200)` renvoie $x = 11.709...$

Activité 3 (Demi-vie et datation au carbone 14).

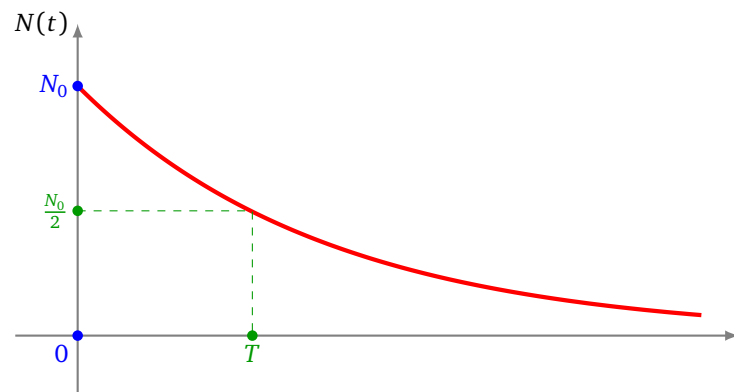
Le carbone 14 est un élément radioactif présent dans le corps de chaque être vivant et qui disparaît peu à peu à sa mort par désintégration. En mesurant le taux de carbone 14 par rapport au taux de carbone ordinaire (qui lui ne se désintègre pas), on peut dater l'époque à laquelle a vécu l'être vivant (jusqu'à 40 000 ans en arrière).

Le nombre d'atomes de carbone 14 suit une loi exponentielle donnée par la formule :

$$N(t) = N_0 \exp\left(-\frac{t \ln(2)}{T}\right)$$

où :

- $N(t)$ est le nombre d'atomes restant après t années,
- N_0 est le nombre d'atomes initial, on prendra ici $N_0 = 1000$,
- T est la période de demi-vie des atomes de carbone 14, $T = 5730$.



1. Programme une fonction `carbone14(t, N0=1000, T=5730)` qui renvoie $N(t)$. Combien reste-t-il d'atomes sur les 1000 atomes de départ au bout de 100 ans ?

2. (a) Vérifie mathématiquement et expérimentalement que

$$N(t) = N_0 \cdot 2^{-t/T}.$$

(b) Combien reste-t-il d'atomes au bout de $T = 5730$ années ? Justifie le terme de « demi-vie » pour la durée T .

(c) Combien reste-t-il d'atomes au bout de $2T$ années ? Au bout de $3T$ années ? ...

(d) Saurais-tu trouver de tête environ combien il reste d'atomes au bout d'une période de 10 demi-vies ?

3. On souhaite dater un échantillon à partir de sa teneur en carbone 14.

(a) Vérifie mathématiquement que

$$t = -\frac{T}{\ln(2)} \ln\left(\frac{N(t)}{N_0}\right).$$

(b) Programme une fonction `datation14(N, N0=1000, T=5730)` qui renvoie la date de l'échantillon en fonction du nombre d'atomes N mesuré.

(c) Vérifie que si on mesure $N = 500$ atomes sur les $N_0 = 1000$ initial, alors l'échantillon a bien l'âge que l'on pense.

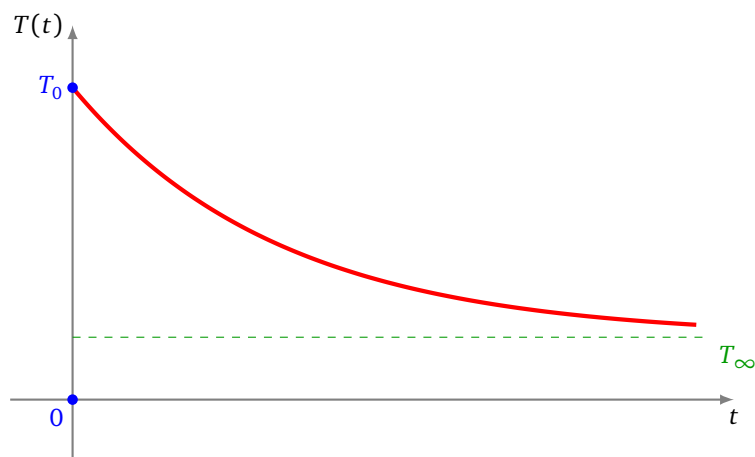
(d) Tu as trouvé un échantillon d'une espèce disparue, l'*Animagus Pythoniscus* avec $N = 200$ atomes sur les $N_0 = 1000$ initial. Quand a vécu cet animal ?

Activité 4 (La loi de refroidissement de Newton).

On place un corps chaud de température initiale T_0 (par exemple $T_0 = 100^\circ\text{C}$) dans une pièce plus froide de température T_∞ (par exemple $T_\infty = 25^\circ\text{C}$). Le corps chaud se refroidit progressivement jusqu'à atteindre la température de la pièce (au bout d'un temps infini). La loi de refroidissement de Newton exprime la température $T(t)$ du corps en fonction du temps t (en minutes) :

$$T(t) - T_\infty = (T_0 - T_\infty)e^{-kt}$$

où k est une constante que l'on va déterminer expérimentalement.



1. Vérifie mathématiquement que $T(0) = T_0$ et que $\lim_{t \rightarrow +\infty} T(t) = T_\infty$.
2. On fixe $T_0 = 100^\circ\text{C}$, $T_\infty = 25^\circ\text{C}$ et on va déterminer k à l'aide d'une information supplémentaire. On mesure qu'à l'instant $t_1 = 10$ minutes, la température du corps est $T_1 = 65^\circ\text{C}$.
Prouve que la constante k est donnée par la formule :

$$k = -\frac{1}{t_1} \ln \left(\frac{T_1 - T_\infty}{T_0 - T_\infty} \right).$$

3. Maintenant que tu connais k , programme une fonction `temperature(t)` qui renvoie la température $T(t)$. Quelle est la température au bout de 20 minutes de refroidissement ?
4. Par tâtonnement, par balayage ou en résolvant une équation, trouve au bout de combien de temps (arrondi à la minute près) la température du corps atteint 30°C .

Activité 5 (Définition de l'exponentielle).

Objectifs : programmer le calcul de $\exp(x)$ par différentes méthodes.

1. **Limite d'une suite.** On a

$$\exp(x) = \lim_{n \rightarrow +\infty} \left(1 + \frac{x}{n} \right)^n.$$

Déduis-en une fonction `exponentielle_limite(x, n)` qui renvoie une valeur approchée de $\exp(x)$ pour une valeur de n (assez grande) fixée.

Teste ta fonction pour calculer $\exp(2.8)$, avec $n = 10$, puis 100... Compare tes résultats avec la fonction Python `exp()`.

2. **Factorielle.** Programme une fonction `factorielle(n)` qui renvoie

$$n! = 1 \times 2 \times 3 \times \cdots \times n.$$

Indications. Le plus simple est d'initialiser une variable `fact` à 1 puis de programmer une boucle. Par convention $0! = 1$. Par exemple $10! = 3\,628\,800$.

3. Somme infinie. On note

$$S_n = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^k}{k!} + \cdots + \frac{x^n}{n!} = \sum_{k=0}^n \frac{x^k}{k!}.$$

Alors

$$\exp(x) = \lim_{n \rightarrow +\infty} S_n.$$

Déduis-en une fonction `exponentielle_somme(x,n)` qui renvoie la valeur de la somme S_n et qui fournit ainsi une valeur approchée de $\exp(x)$.

Teste ta fonction avec $n = 10$, $n = 15$...

4. Méthode de Hörner. Afin de minimiser les multiplications (du genre $x^k = x \times x \times x \cdots$) voici la formule de Hörner qui est juste une réécriture de la somme S_n définie à la question précédente :

$$S_n = 1 + \frac{x}{1} \left(1 + \frac{x}{2} \left(1 + \frac{x}{3} \left(\cdots + \frac{x}{n-1} \left(1 + \frac{x}{n} \right) \right) \right) \right)$$

Et bien sûr :

$$\exp(x) = \lim_{n \rightarrow +\infty} S_n.$$

Programme une fonction `exponentielle_horner(x,n)` qui implémente cette méthode et renvoie la valeur de S_n .

Indications. Il faut partir du terme le plus imbriqué $1 + \frac{x}{n}$ puis construire cette expression à rebours à l'aide d'une boucle.

5. (Un peu de théorie plus difficile.) Compare le nombre de multiplications effectuées pour les méthodes des deux questions précédentes pour le calcul de S_n . Par exemple le calcul de $\frac{x^3}{3!}$ nécessite deux multiplications pour $x^3 = x \times x \times x$ et deux multiplications pour $3! = 1 \times 2 \times 3$. (Note : on ne compte pas les additions car c'est une opération peu coûteuse, et ici on ne tient pas compte des divisions car il y en a autant pour les deux méthodes.)

6. Fraction continue d'Euler. Voici une nouvelle formule pour $S_n = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}$ sous la forme d'une succession de fractions :

$$S_n = \frac{1}{1 - \frac{x}{1 + x - \frac{x}{2 + x - \frac{x}{3 + x - \frac{x}{4 + x - \cdots \frac{(n-1)x}{n-1 + x - \frac{x}{n + x}}}}}}}$$

On programme cette formule en partant de la fraction tout en bas par l'algorithme suivant :

Algorithme.

- Action : calculer la somme S_n en fonction de x .
- Initialiser $S \leftarrow 0$.
- Pour k allant de n à 1 (donc à rebours), faire :

$$S \leftarrow \frac{x}{k + x - kS}$$

- À la fin, faire $S \leftarrow \frac{1}{1-S}$.
- Renvoyer S .

Programme cet algorithme en une fonction `exponentielle_euler(x,n)`.

7. **Exponentielle de grandes valeurs.** Les fonctions précédentes sont valables quel que soit x , mais pour de grandes valeurs de x (par exemple $x = 100.5$) il faut de grandes valeurs de n pour obtenir une bonne approximation de $\exp(x)$. Pour remédier à ce problème nous allons voir un algorithme qui permet de se ramener au calcul de l'exponentielle d'un réel $f \in [0, 1[$ pour lequel les fonctions précédentes sont efficaces.

L'idée est de décomposer x en sa partie entière plus sa partie fractionnaire :

$$x = k + f \quad \text{où } k \text{ est un entier et } 0 \leq f < 1.$$

On utilise la propriété de l'exponentielle :

$$e^x = e^{k+f} = e^k \times e^f.$$

Maintenant :

- $e^f = \exp(f)$ s'obtient par le calcul de l'exponentielle d'un petit réel $0 \leq f < 1$ et se calcule bien par l'une des méthodes précédentes.
- $e^k = e \times e \times \dots \times e$ est le produit de plusieurs e , c'est donc un simple calcul de puissance (et pas vraiment un calcul d'exponentielle).
- Il faut au préalable avoir calculé une fois pour toutes la valeur de la constante $e = \exp(1) = 2.718\dots$ par l'une des méthodes précédentes.

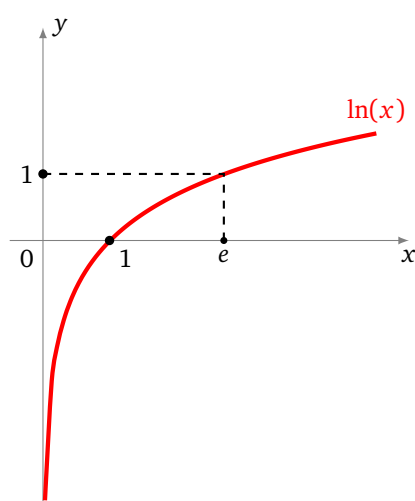
Voici l'algorithme à programmer en une fonction `exponentielle_astuce(x,n)` :

Algorithme.

- Action : calculer une approximation de $\exp(x)$.
- Préalable : calculer une fois pour toutes la valeur de $e = \exp(1)$ avec le maximum de précision.
- Poser k la partie entière de x (utiliser `floor()`).
- Poser $f = x - k$.
- Calculer une valeur approchée de $\exp(f)$ par l'une des méthodes précédentes en fonction d'un paramètre n .
- Calculer $\exp(k) = e^k$ par le calcul de puissance $e \times e \times \dots$.
- Renvoyer l'approximation correspondant au résultat $\exp(x) = \exp(k) \times \exp(f)$.

Logarithme

Le logarithme est une fonction aussi importante que l'exponentielle. C'est le logarithme qui donne l'ordre de grandeur de certaines quantités physiques, par exemple la puissance d'un séisme ou celle d'un son.



Cours 1 (Le logarithme décimal).

On commence avec le logarithme décimal qui est plus facile à appréhender. Le logarithme décimal d'un nombre réel positif x , est l'exposant y de ce nombre écrit sous la forme $x = 10^y$. Autrement dit :

$$x = 10^y \iff y = \log_{10}(x)$$

Exemples.

- $\log_{10}(10^2) = 2$, $\log_{10}(10^3) = 3$, $\log_{10}(10\,000) = 4$,...
- On a aussi $\log_{10}(10) = 1$, $\log_{10}(1) = 0$.
- Comme $0.1 = \frac{1}{10} = 10^{-1}$, on a $\log_{10}(0.1) = -1$.
- Le logarithme est défini pour n'importe quel $x > 0$. Par exemple pour $x = 25.5$, on a $\log_{10}(x) = 1.4065\dots$. Ce qui signifie que $10^{1.4065\dots} = 25.5$.

Propriété. La propriété fondamentale du logarithme est :

$$\log_{10}(a \times b) = \log_{10}(a) + \log_{10}(b)$$

Par exemple $a = 10^2$, $b = 10^3$, on a $a \times b = 10^2 \times 10^3 = 10^{2+3} = 10^5$. On a bien

$$\log_{10}(a) + \log_{10}(b) = \log_{10}(10^2) + \log_{10}(10^3) = 2 + 3 = 5 = \log_{10}(10^5) = \log_{10}(a \times b)$$

Cours 2 (Le(s) logarithme(s) avec Python).

Avertissement : il y a un conflit entre mathématiciens et informaticiens pour la notation du logarithme !

- **Logarithme décimal.**

- Notation mathématique : $\log_{10}(x)$
- Commande Python : `log(x, 10)`

- **Logarithme népérien.**

- Notation mathématique : $\ln(x)$
- Commande Python : `log(x)`

- **Logarithme en une autre base.**

- Notation mathématique : $\log_b(x)$
- Commande Python : `log(x, b)`

Exemple : avec $x = 25.5$, alors on calcule $\log_{10}(x)$ par la commande `log(25.5, 10)` qui renvoie

$$\log_{10}(x) \simeq 1.406540180433955$$

On vérifie le résultat en calculant 10^y , où $y = 1.4065\dots$ par la commande `10**y` qui renvoie :

$$25.499999999999999$$

Bien sûr, tous les calculs effectués par Python avec les nombres flottants sont des calculs approchés.

Activité 1 (Logarithme décimal - Échelle de Richter).

Objectifs : comprendre l'échelle de Richter qui mesure la force d'un tremblement de terre.

On quantifie la force d'un séisme par un nombre, appelé **magnitude**, qui dépend de la puissance délivrée par une secousse :

$$M = \frac{2}{3} \log_{10} \left(\frac{E}{E_0} \right) - 3.2$$

où :

- $\log_{10}$ est le logarithme décimal,
- E est l'énergie délivrée par le séisme (exprimée en joules),
- $E_0 = 1.6 \times 10^{-5}$ joules est une énergie de référence.

Description	Magnitude	Effets	Fréquence moyenne
Micro	moins de 1.9	Micro tremblement de terre, non ressenti.	8 000 par jour
Très mineur	2.0 à 2.9	Généralement non ressenti mais détecté/enregistré.	1 000 par jour
Mineur	3.0 à 3.9	Souvent ressenti sans causer de dommages.	50 000 par an
Léger	4.0 à 4.9	Secousses notables d'objets à l'intérieur des maisons, bruits d'entrechoquement. Les dommages restent très légers.	6 000 par an
Modéré	5.0 à 5.9	Peut causer des dommages significatifs à des édifices mal conçus dans des zones restreintes. Pas de dommages aux édifices bien construits.	800 par an
Fort	6.0 à 6.9	Peut provoquer des dommages sérieux sur plusieurs dizaines de kilomètres. Seuls les édifices adaptés résistent près du centre.	120 par an
Très fort	7.0 à 7.9	Peut provoquer des dommages sévères dans de vastes zones ; tous les édifices sont touchés près du centre.	18 par an
Majeur	8.0 à 8.9	Peut causer des dommages très sévères dans des zones à des centaines de kilomètres à la ronde. Dommages majeurs sur tous les édifices, y compris à des dizaines de kilomètres du centre.	1 par an
Dévastateur	9.0 et plus	Dévaste des zones sur des centaines de kilomètres à la ronde. Dommages sur plus de 1 000 kilomètres à la ronde.	1 à 5 par siècle

Source : « Magnitude (Sismologie) » Wikipédia.

1. Programme une fonction $\text{magnitude}(E)$ qui renvoie la magnitude d'un séisme dont l'énergie E est donnée.
Exemple. Vérifie qu'un séisme libérant une énergie $E_1 = 10^6$ joules est de magnitude 4.
2. Pour des énergies de la forme $E = 10^i$, calcule la magnitude correspondante jusqu'à obtenir un séisme de magnitude supérieure à 9.
3. Par tâtonnement, balayage ou en résolvant une équation, trouve l'énergie environ nécessaire pour obtenir un séisme de magnitude 7.
4. Vérifie expérimentalement, puis montre mathématiquement, que si $E_2 = 1000E_1$ alors $M_2 = M_1 + 2$ (quelle que soit l'énergie E_1). Trouve expérimentalement (ou mathématiquement) quel facteur k , avec $E_2 = kE_1$ permet d'obtenir $M_2 = M_1 + 1$ (quelle que soit l'énergie E_1).

Activité 2 (Logarithme décimal - Décibels).

Objectifs : calculer le niveau sonore.

On mesure le niveau de bruit en décibels (dB) qui correspond à la puissance d'un son (par rapport à une puissance de référence). La formule est

$$D = 20 \log_{10} \left(\frac{P}{P_0} \right)$$

où :

- $\log_{10}$ est le logarithme décimal,
- P est la pression mesurée du son (exprimée en pascal Pa),
- $P_0 = 2 \times 10^{-5}$ Pa est une pression de référence.

1. Programme une fonction `decibels(P)` qui renvoie le niveau de bruit d'un son de puissance P donnée.

Exemple. Vérifie qu'une puissance $P = 1$ Pa, correspond à $D = 94$ décibels.

2. Complète le tableau suivant :

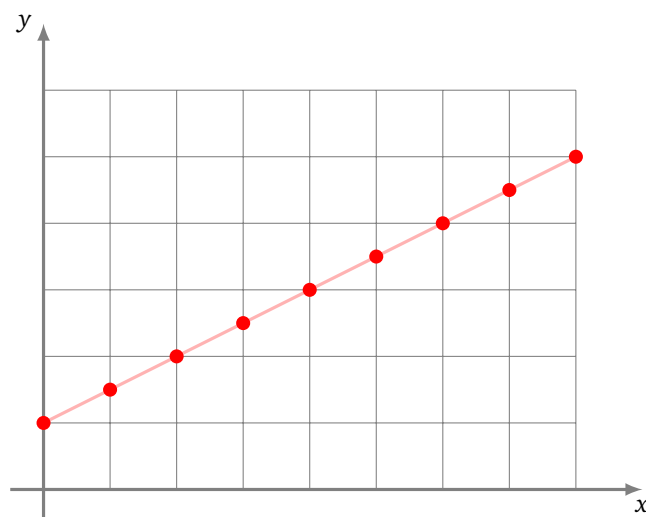
Bruit	Pression (Pa)	Niveau (dB)
Moteur d'avion à réaction (à 1 mètre)	632	
Marteau-piqueur (à 1 mètre)	2	
Niveau de dommage à l'oreille	$P > 0.355$	
Niveau de gêne		$D > 70$
Conversation (à 1 mètre)	0.002 à 0.02	
Chambre calme		10 à 20
Seuil de l'audition à 1kHz (à l'oreille)	$2 \cdot 10^{-5}$	
Chambre anéchoïque		-10

Source : « Sound pressure » Wikipédia.

Cours 3 (Échelle logarithmique).

Relation $y = ax + b$. On considère des données du type (x_i, y_i) et on veut étudier le lien entre y_i et x_i . On détecte facilement une relation affine du type $y = ax + b$ en plaçant les points sur un graphique. Une telle relation existe si et seulement si les points sont tous sur une même droite.

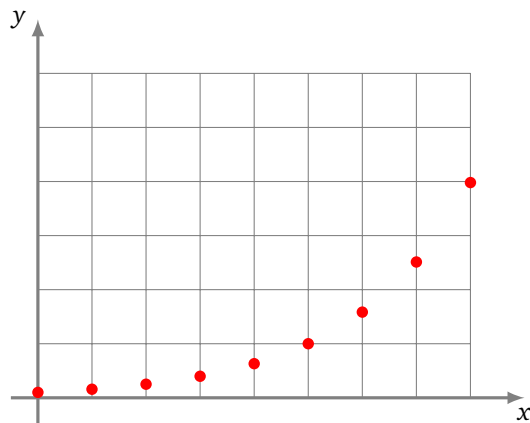
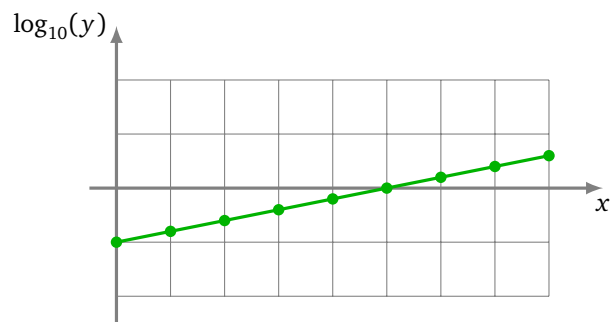
Ci-dessous des points vérifiant la relation $y = \frac{1}{2}x + 1$.



Points de coordonnées (x, y)

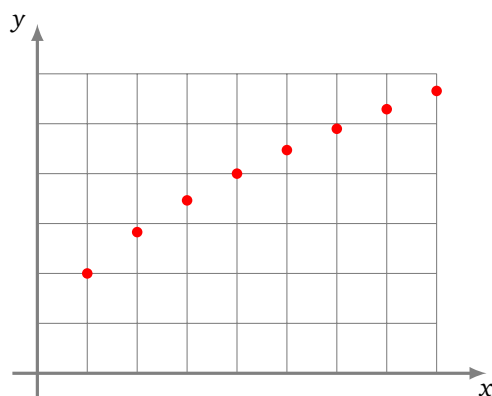
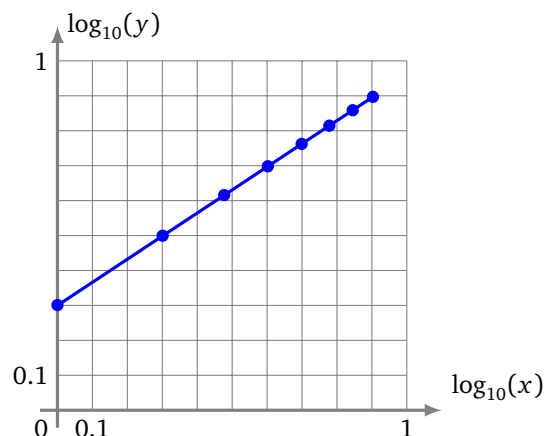
Relation $y = 10^{ax+b}$. Les données sont du type (x_i, y_i) mais cette fois la relation est du type $y = 10^{ax+b}$. Si on trace les points directement sous la forme (x, y) on ne voit rien de spécial (figure ci-dessous à gauche, les points rouges). Par contre si on place les points $(x, \log_{10}(y))$ alors les points sont alignés (figure ci-dessous à droite, les points verts). (Preuve : comme $y = 10^{ax+b}$ alors $\log_{10}(y) = ax + b$.)

Ci-dessous des points vérifiant la relation $y = 10^{\frac{1}{5}x-1}$.

Points de coordonnées (x, y) Points de coordonnées $(x, \log_{10}(y))$

Relation $y = bx^a$. Les données sont du type (x_i, y_i) avec la relation $y = bx^a$. Le tracé des points (x, y) ne donne rien (figure ci-dessous à gauche, les points rouges). Par contre le tracé des points $(\log_{10}(x), \log_{10}(y))$ donne des points alignés (figure ci-dessous à droite, les points bleus). (Preuve : comme $y = bx^a$ alors $\log_{10}(y) = \log_{10}(bx^a)$ donc $\log_{10}(y) = \log_{10}(b) + \log_{10}(x^a)$, d'où $\log_{10}(y) = a \log_{10}(x) + \log_{10}(b)$. Si on pose $Y = \log_{10}(y)$ et $X = \log_{10}(x)$ on trouve une relation affine $Y = aX + \log_{10}(b)$.)

Ci-dessous des points vérifiant la relation $y = 2x^{\frac{1}{2}}$ (c'est-à-dire $y = 2\sqrt{x}$).

Points de coordonnées (x, y) Points de coordonnées $(\log_{10}(x), \log_{10}(y))$

Activité 3 (Le logarithme décimal - Échelle logarithmique).

Objectifs : utiliser le logarithme pour détecter des comportements particuliers.

- Programme une fonction `afficher_points_xy(points)` qui affiche (en rouge) chaque point de coordonnées (x, y) à partir d'une liste de points.
 - Programme une fonction `afficher_points_xlogy(points)` qui affiche (en vert) chaque point de coordonnées $(x, \log_{10}(y))$ à partir d'une liste de points (x, y) donnée.
 - Programme une fonction `afficher_points_logxlogy(points)` qui affiche (en bleu) chaque point de coordonnées $(\log_{10}(x), \log_{10}(y))$ à partir d'une liste de points (x, y) donnée.
- Voici trois séries de données (x, y) :

x	y
2	5.66
3	10.39
5	22.36
7	37.04
11	72.97

x	y
2	5
3	6.5
5	9.5
7	12.5
11	18.5

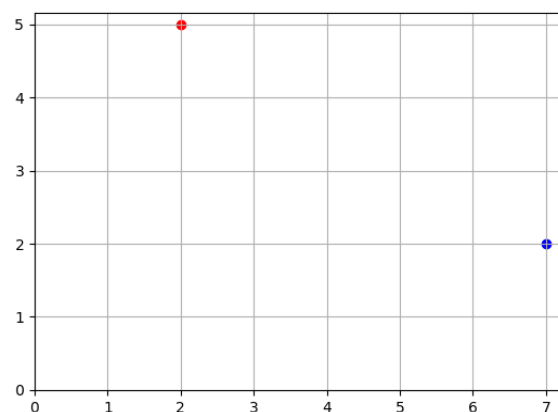
x	y
2	5.01
3	6.31
5	10.00
7	15.84
11	39.81

Reconnais par affichage graphique, celle qui est de la forme $y = ax + b$, celle qui est de la forme $y = 10^{ax+b}$ et celle de la forme $y = bx^a$.

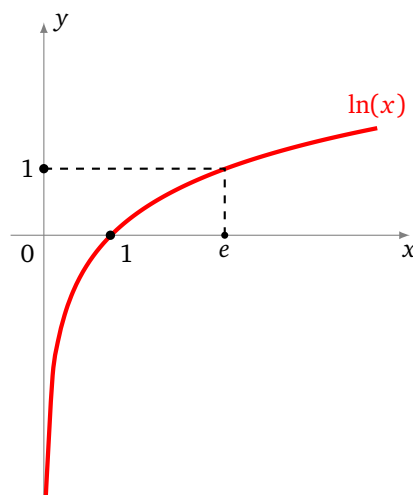
Bonus. Calcule les constantes a et b qui conviennent dans chacun des cas.

Utiliser Matplotlib. Voici un bref programme qui affiche un point rouge de coordonnées (2, 5) et un point bleu de coordonnées (7, 2).

```
import matplotlib.pyplot as plt
plt.scatter(2,5,color="red")
plt.scatter(7,2,color="blue")
plt.axes().set_aspect('equal')
plt.xlim(xmin=0)
plt.ylim(ymin=0)
plt.grid()
plt.show()
```



Cours 4 (Logarithme népérien).



- La **fonction logarithme népérien** est la fonction $\ln :]0, +\infty[\rightarrow \mathbb{R}$ qui vérifie :

$$\ln(1) = 0 \quad \text{et} \quad \ln(x \times y) = \ln(x) + \ln(y)$$

- La fonction logarithme est strictement croissante, $\lim_{x \rightarrow 0^+} \ln(x) = -\infty$, $\lim_{x \rightarrow +\infty} \ln(x) = +\infty$.
- $\ln(1/x) = -\ln(x)$, $\ln(x^n) = n \ln(x)$.
- La dérivée du logarithme est : $\ln'(x) = \frac{1}{x}$.

- La fonction logarithme $\ln :]0, +\infty[\rightarrow \mathbb{R}$ est la bijection réciproque de la fonction exponentielle $\exp : \mathbb{R} \rightarrow]0, +\infty[$, c'est-à-dire :

$$y = \ln(x) \iff x = \exp(y)$$

Plus précisément :

$$\begin{array}{ll} \exp(\ln(x)) & \text{pour tout } x > 0, \\ \ln(\exp(x)) & \text{pour tout } x \in \mathbb{R}. \end{array}$$

- On note $e = \exp(1) = 2.718\dots$ et alors $\ln(e) = 1$.
- Le logarithme et l'exponentielle permettent de définir une puissance avec des exposants réels : $a^b = \exp(b \ln(a))$.

Cours 5 (Tables de logarithmes).

Le logarithme a été introduit au début des années 1600 pour effectuer facilement des multiplications à plusieurs chiffres nécessaires aux calculs astronomiques.

[illegible]

« *Arithmetica logarithmica* » Tables de logarithmes de H. Briggs, 1624.

Tables de logarithmes.

Le préalable est de calculer une table des logarithmes, c'est-à-dire les valeurs approchées de $\ln(x)$ pour plein de valeurs de x . Par exemple, voici le début d'une table de logarithme avec 4 décimales après la

virgule :

x	$y = \ln(x)$
1.000	0.0000
1.001	0.0010
...	...
1.123	0.1160
1.124	0.1169
...	...
2.000	0.6931
2.001	0.6936
2.002	0.6941
...	...

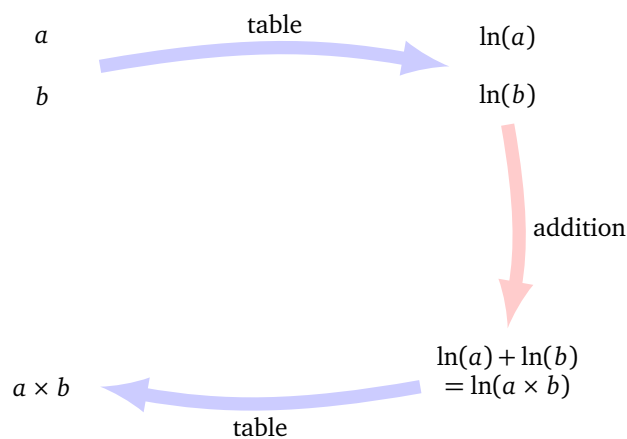
x	$y = \ln(x)$
...	...
2.567	0.9427
2.568	0.9431
...	...
2.718	0.9999
2.719	1.0003
...	...
2.884	1.0592
2.885	1.0595
2.886	1.0599
...	...

Lecture de la table.

- Pour chercher le logarithme d'un nombre x il suffit de consulter la table de la gauche vers la droite. Par exemple, on lit que pour $x = 1.123$ on a $\ln(x) \simeq 0.1160$.
- L'opération inverse est tout aussi utile, étant donné un nombre y , trouver le réel x tel que $\ln(x) = y$. Cela revient à calculer $x = \exp(y)$! Pour cela on lit la table de droite à gauche. Par exemple quelle est l'exponentielle de $y = 0.6931$? C'est environ $x = 2.000$.

Multiplications faciles.

Voici le principe pour calculer $a \times b$ sans efforts.



On voit qu'il suffit de faire une addition et trois recherches dans la table.

Exemple. $a = 1.124$ et $b = 2.567$. On veut calculer $a \times b$. On cherche $\ln(a)$ dans la table, on trouve $\ln(a) \simeq 0.1169$, puis $\ln(b) \simeq 0.9427$. On calcule $\ln(a) + \ln(b) \simeq 1.0596$. On a donc $\ln(a \times b) \simeq 1.0596$. On cherche dans la table quel nombre x correspond à un logarithme $y = 1.0596$. L'entrée qui correspond le mieux est $c = 2.885$. Bilan : $a \times b \simeq 2.885$.

Remarques historiques.

- H. Briggs a calculé les tables de logarithmes pour 30 000 entrées avec 14 décimales pour chaque logarithme.
- Les tables calculées étaient souvent les tables du logarithme décimal $\log_{10}$. L'avantage est le suivant : une fois que vous avez la table de $\log_{10}$ pour $x = 1.001$, $x = 1.002$, ..., $x = 9.999$, $x = 10.000$, alors

vous savez calculer le logarithme décimal de n'importe quel nombre. Par exemple comment calculer le logarithme décimal de $x = 574.5$? Il suffit de décaler la virgule :

$$\log_{10}(574.5) = \log_{10}(100 \times 5.745) = \log_{10}(100) + \log_{10}(5.745) = 2 + \log_{10}(5.745)$$

Il ne reste plus qu'à consulter la table pour connaître $\log_{10}(5.745)$.

Activité 4 (Logarithme népérien).

Objectifs : utiliser les propriétés du logarithme népérien pour faire des multiplications sans efforts.

1. **Propriétés du logarithme.** Vérifie expérimentalement avec Python les propriétés du logarithme :

$$\ln(a \times b) = \ln(a) + \ln(b) \quad \ln(1/a) = -\ln(a)$$

$$\ln(a/b) = \ln(a) - \ln(b) \quad \ln(a^n) = n \ln(a)$$

$$\ln(\sqrt{a}) = \frac{1}{2} \ln(a) \quad a^b = \exp(b \ln(a))$$

Prends par exemple $a = 2$, $b = 3$, $n = 7$, puis $a = 3/2$, $b = 1/3$, $n = \pi$.

Vérifie aussi expérimentalement que $\lim_{x \rightarrow 0^+} \ln(x) = -\infty$, $\ln(1) = 0$, $\ln(e) = 1$. Convains-toi expérimentalement que $\ln(e^n) = n$ et que l'on a $\lim_{x \rightarrow +\infty} \ln(x) = +\infty$.

2. **Tables simulées.** Programme une fonction `table_ln(x, N)` et une fonction `table_exp(x, N)` qui renvoie la valeur du logarithme (ou de l'exponentielle) en x , tronquée à N chiffres après la virgule. Ces deux fonctions jouent le rôle de la consultation des tables de logarithmes à N décimales.

Exemple. Avec $x = 54$ et $N = 4$, alors $\ln(x) = 3.988984046 \dots$ et `table_ln(x, N)` renvoie 3.9889.

Indications. Étant donnés x et N (ex. $x = 12.3456789$ et $N = 2$) :

- on peut multiplier par une puissance de 10 pour décaler la virgule (ex. $x \times 100 = 1234.6789$),
- puis prendre la partie entière ($E(1234.56789) = 1234$),
- puis décaler la virgule cette fois vers la droite en divisant par la même puissance de 10 ($1234/100 = 12.34$).

3. **Multiplication par les tables.** Programme une fonction `multiplication(a, b, N)` qui renvoie une valeur approchée de $a \times b$ sans faire directement de multiplication, mais en consultant les tables :

- cherche dans la table une valeur approchée de $\ln(a)$ et $\ln(b)$,
- calcule $\gamma = \ln(a) + \ln(b)$,
- cherche dans la table une valeur approchée de $\exp(\gamma) = a \times b$.

On a bien remplacé une multiplication, par une addition.

Exemple. Calcule 98.765×43.201 . Combien doit valoir N pour obtenir une valeur approchée du produit avec 3 chiffres exacts après la virgule ?

Cours 6 (Logarithme en base quelconque).

Soit b un réel positif. Le **logarithme en base b** est défini par

$$\log_b(x) = \frac{\ln(x)}{\ln(b)}$$

Par exemple

$$\log_7(49) = \frac{\ln(49)}{\ln(7)} = \frac{\ln(7^2)}{\ln(7)} = \frac{2 \ln(7)}{\ln(7)} = 2$$

- **Logarithme décimal.** Si $b = 10$, on a la formule $\log_{10}(x) = \frac{\ln(x)}{\ln(10)}$.

- *Logarithme népérien.* Si $b = e$, on a $\log_e(x) = \frac{\ln(x)}{\ln(e)} = \ln(x)$.
- *Logarithme en base 2.* Il est particulièrement utile en informatique ! Avec $b = 2$, on a $\log_2(x) = \frac{\ln(x)}{\ln(2)}$. Il vérifie que $\log_2(2^k) = k$.
Exemple. On peut coder $n = 256$ entiers (de 0 à 255) sur $k = 8$ bits. Quel est le lien entre n et k ? On a $\log_2(256) = \log_2(2^8) = 8$, c'est-à-dire $k = \log_2(n)$.

Activité 5 (Logarithme en base quelconque).

Objectifs : travailler avec des logarithmes dans d'autres bases.

1. **Logarithme entier en base 10.** Le **logarithme entier en base 10** est le plus grand entier k tel que $10^k \leq x$. Programme une boucle « tant que » qui renvoie cet entier k . Vérifie que c'est aussi la partie entière de $\log_{10}(x)$.
Indication. Attention au décalage !
Exemple. Avec $x = 666$, alors $10^2 = 100 \leq x < 1000 = 10^3$ donc le logarithme entier de x en base 10 vaut $\ell = 2$. Par ailleurs $\log_{10}(x) = 2.823 \dots$ dont la partie entière est bien 2.
2. **Logarithme entier en base 2.** Fais le même travail avec le **logarithme entier en base 2** qui est le plus grand entier k tel que $2^k \leq x$. Vérifie que c'est bien la partie entière de $\log_2(x)$.
Exemple. Avec $x = 666$, alors $2^9 = 512 \leq x < 1024 = 2^{10}$. Donc le logarithme entier de x en base 2 vaut $\ell = 9$. Par ailleurs $\log_2(x) = 9.379 \dots$ dont la partie entière est bien 9.
3. **Dichotomie.** Voici une variante du jeu de la devinette. Il s'agit de trouver un entier k parmi les entiers de $[0, n[$. On propose une réponse i , et on obtient la réponse « intervalle de gauche $[0, i[$ » ou « intervalle de droite $[i, n[$ ». On gagne quand on a obtenu un intervalle ne contenant qu'un élément. Pour optimiser mes chances je décide de couper l'intervalle en deux à chaque fois.

Question. Au bout de combien d'étapes suis-je certain de trouver l'entier k ?

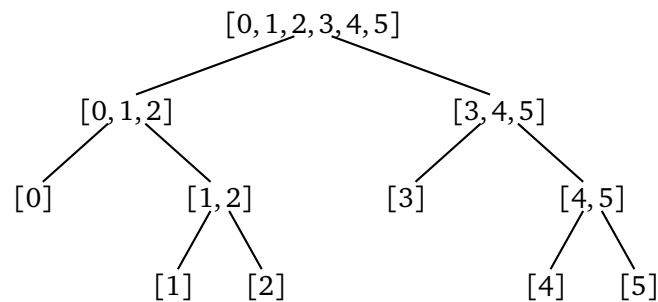
Travail à faire. Programme une fonction `dichotomie(n)` qui renvoie cet entier k dans le pire des cas.

Indications. Il ne s'agit pas vraiment de programmer le jeu mais seulement d'étudier le pire des cas. Pars d'un intervalle d'entiers $[0, n[$. Divise à chaque étape l'intervalle en deux sous-intervalles en coupant au milieu (de rang $n//2$). Attention une partie peut être plus grande que l'autre si n est impair. Garde l'intervalle le plus grand. Continue tant que la longueur de cet intervalle est strictement supérieure à 1. Compte le nombre de découpages effectués.

Exemple. $n = 6$ et l'entier à trouver est $k = 4$. Les entiers possibles sont donc dans $[0, 5]$.

- Je propose $i = n//6 = 3$. On me répond : « l'entier à trouver est dans l'intervalle de droite $[3, 5]$ ».
 - L'intervalle $[3, 5]$ est de longueur $n' = 3$, je découpe au rang $n'/2 = 1$ donc en deux sous-intervalles $[3]$ et $[4, 5]$. On me répond : « l'entier à trouver est dans l'intervalle de droite $[4, 5]$ ».
 - L'intervalle $[4, 5]$ est de longueur $n'' = 2$, je découpe au rang $n''/2 = 1$ donc en deux sous-intervalles $[4]$ et $[5]$. On me répond : « l'entier à trouver est dans l'intervalle de gauche $[4]$ ».
- Comme c'est un intervalle qui ne contient qu'un seul entier, j'ai gagné. Il m'a fallu 3 étapes.

Arbre. Voici le schéma de toutes les configurations possibles avec la méthode de la dichotomie, pour $n = 6$. Certains entiers (0 et 3) sont trouvés en 2 étapes. Les autres nécessitent 3 étapes.



Réponse. Pour n donné, compare ta réponse avec :

- $\log_2(n)$, le logarithme de n en base 2,
- le logarithme entier de n en base 2.

Commence par le cas où n est une puissance de 2.

Combien faut-il d'étapes au maximum pour déterminer un entier entre 0 et 1000 ?

4. **Logarithme en base quelconque.** Programme une fonction `logarithme_base(x, b)` qui renvoie le logarithme de x en base b selon la formule :

$$\log_b(x) = \frac{\ln(x)}{\ln(b)}.$$

Vérifie ta fonction en comparant ton résultat avec la commande Python `log(x, b)`. En prenant $b = 10$ on obtient $\log_{10}$, le logarithme décimal. Quelle valeur de la base b , donne le logarithme népérien $\ln$?

5. **Nombre de chiffres dans une base quelconque.**

- Le nombre de chiffres de l'écriture décimale d'un entier n est l'entier k tel que $10^{k-1} < n \leq 10^k$. Autrement dit c'est $k = E(\log_{10}(x)) + 1$ (où $E(x)$ désigne la partie entière d'un réel x).
- Le nombre de chiffres de l'écriture binaire d'un entier n est l'entier k tel que $2^{k-1} < n \leq 2^k$. Autrement dit c'est $k = E(\log_2(x)) + 1$.
- Plus généralement, le nombre de chiffres de l'écriture en base b d'un entier n est l'entier k tel que $b^{k-1} < n \leq b^k$. Autrement dit c'est $k = E(\log_b(x)) + 1$.

Exemple. Prenons $n = 123$.

- En base 10, on a $10^2 < 123 \leq 10^3$, le nombre de chiffres est bien $k = 3$ et $\log_{10}(x) = 2.089\dots$, on retrouve bien $k = E(\log_{10}(x)) + 1 = 2 + 1 = 3$.
- En base 2, on a $64 = 2^6 < 123 \leq 2^7 = 128$, il faut donc $k = 7$ chiffres. Ce qui se vérifie aussi par $k = E(\log_2(123)) + 1 = E(6.942\dots) + 1 = 7$. Enfin la commande `bin(123)` renvoie '0b1111011' l'écriture binaire de n est donc 1.1.1.1.0.1.1 et nécessite 7 chiffres.
- En base 16, on a $16^1 < 123 \leq 16^2 = 256$, il faut donc $k = 2$ chiffres. Ce qui se vérifie aussi par $k = E(\log_{16}(123)) + 1 = E(1.735\dots) + 1 = 2$. Enfin la commande `hex(123)` renvoie '0x7b' l'écriture hexadécimale de n est donc 7.B et nécessite 2 chiffres.

Programme une fonction `nombre_de_chiffres(n, b)` qui renvoie le nombre de chiffres nécessaires à l'écriture de l'entier n en base b .

Vérifie tes résultats en base 2 à l'aide de `bin()` et en base 16 avec `hex()`.

Activité 6 (Calcul du logarithme I).

Objectifs : utiliser des formules qui permettent de calculer nous-même le logarithme.

1. Logarithme par série (1).

On a la formule :

$$\ln(1+u) = u - \frac{u^2}{2} + \frac{u^3}{3} - \cdots + (-1)^{k-1} \frac{u^k}{k} + \cdots$$

En posant $x = 1 + u$ (et donc $u = x - 1$) cela permet de calculer $\ln(x)$. Attention cette formule n'est valable que pour $u \in]-1, +1[$ c'est-à-dire pour x proche de 1.

Programme une fonction `logarithme_serie_1(x, N)` qui pour $x \in]0, 2[$, renvoie la valeur approchée de $\ln(x)$ en calculant la somme de termes $(-1)^{k-1} \frac{u^k}{k}$, pour $k < N$.

Indications.

- Commence par poser $u = x - 1$, puis calcule la somme.
- Le terme $(-1)^{k-1}$ vaut -1 si k est pair et $+1$ si k est impair.

Pour $x = 1.543$ et $N = 10$, quelle approximation de $\ln(x)$ obtiens-tu ? Compare avec la fonction Python.

2. Logarithme par série (2).

On a la formule :

$$\ln\left(\frac{1+u}{1-u}\right) = 2u + 2\frac{u^3}{3} + 2\frac{u^5}{5} + \cdots$$

valable pour $u \in]-1, +1[$. Déduis-en une fonction `logarithme_serie_2(x, N)` qui pour $x \in]0, 2[$, renvoie la valeur approchée de $\ln(x)$ avec des termes ne dépassant pas le degré N .

Indications.

- Vérifie que si on pose $x = \frac{1+u}{1-u}$ alors $u = \frac{x-1}{x+1}$.
- Calcule une somme de termes $2\frac{u^k}{k}$ pour k parcourant la liste donnée par `range(1, N, 2)`.

Pour $x = 1.543$ et $N = 10$, quelle approximation de $\ln(x)$ obtiens-tu ? Compare avec la fonction précédente et la fonction Python.

3. Réduction d'intervalle.

Les deux formules précédentes sont valables pour x proche de 1 (en fait $0 < x < 2$). Pour obtenir le logarithme d'un nombre $x > 0$ quelconque, il faut se ramener dans l'intervalle $]0, 2[$.

On a la propriété suivante, pour chaque $x > 0$ il existe un réel y avec $0.5 < y < 1.5$ et un entier $k \in \mathbb{Z}$ tel que :

$$x = ye^k$$

où $e = \exp(1)$. Par les propriétés du logarithme, montre que :

$$\ln(x) = \ln(y) + k.$$

Programme une fonction `reduction_intervalle_e(x)` qui renvoie la valeur y et l'entier k demandés.

Exemple. Avec $x = 10$, on écrit $x = \frac{10}{e^2} \cdot e^2$ avec $y = \frac{10}{e^2} = 1.35\dots$ et $k = 2$.

Indications. Tant que $x > 1.5$ alors divise x par e et chaque fois incrémente la valeur de k . Il faut ensuite aussi considérer le cas où $x < 0.5$.

4. Logarithme par série (3).

Programme une fonction `logarithme_serie_3(x, N)` qui renvoie une valeur approchée de $\ln(x)$ quel que soit $x > 0$.

Indications.

- Commence par te ramener à $y \in]0.5, 1.5[$ par la fonction `reduction_intervalle_e(x)` qui renvoie une valeur y et k .
- Calcule $\ln(y)$ par une de tes fonctions précédentes.
- Puis utilise la formule $\ln(x) = \ln(y) + k$.

Pour $x = 154.3$ et $N = 10$, quelle approximation de $\ln(x)$ obtiens-tu ?

Même si les formules de cette activité sont efficaces, ce n'est pas comme cela que les ordinateurs calculent les logarithmes !

Activité 7 (Calcul du logarithme II).

Objectifs : étudier des algorithmes encore plus efficaces pour calculer les logarithmes.

1. Logarithme comme réciproque de l'exponentielle.

On sait calculer la valeur de l'exponentielle (voir la fiche « Exponentielle »). Le logarithme est la bijection réciproque de l'exponentielle, autrement dit :

$$\exp(x) = y \iff y = \ln(x)$$

Pour calculer $\ln(x)$ on procède ainsi :

- On fixe $x > 0$.
- On résout l'équation d'inconnue y : « $\exp(y) = x$ ».

Pour résoudre l'équation $\exp(y) = x$ (d'inconnue y) on utilise par exemple la méthode de Newton pour trouver le zéro de la fonction $f(y) = \exp(y) - x$ (voir la fiche « Dérivée »). Ce qui donne dans la pratique :

- Fixer $x > 0$.
- Définir $u_0 = 1$.
- Puis par récurrence $u_{n+1} = u_n - \frac{\exp(u_n) - x}{\exp(u_n)}$.
- La suite $(u_n)_{n \in \mathbb{N}}$ tend vers $y = \ln(x)$.

Programme cette méthode en une fonction `logarithme_inverse(x, N)` qui renvoie le terme u_N de la suite comme valeur approchée de $\ln(x)$. Compare avec les méthodes de l'activité précédente.

2. Réduction d'intervalle.

Pour $x > 0$ il existe un réel y tel que $1 \leq y < 10$ et un entier $k \in \mathbb{Z}$ tel que

$$x = y \cdot 10^k$$

Programme une fonction `reduction_intervalle_10(x)` qui renvoie ce y et ce k .

Exemple. $x = 617.4 = 6.174 \times 100 = 6.174 \times 10^2$, donc $y = 6.174$ et $k = 2$.

Indication. Base-toi sur le modèle de la fonction `reduction_intervalle_e(x)` de l'activité précédente.

3. Algorithme Cordic.

C'est cet algorithme qui est implémenté dans les calculatrices et utilise des puissances de 10 pour calculer $\ln(x)$. Pour les ordinateurs c'est la version en base 2 qui est préférée.

Programme l'algorithme suivant en une fonction `logarithme_cordic(x, N)`. Pour $x = 1.543$ et $N = 10$ quelle approximation de $\ln(x)$ obtiens-tu ? Compare avec les fonctions précédentes.

Algorithme.

- Entrée : un nombre $x > 0$, un nombre d'itérations N .
- Sortie : une approximation de $\ln(x)$.
- Préalable : calculer une fois pour toute la valeur de $\ln(10)$ et les valeurs $\ln(1 + 10^{-i})$ pour i variant de 0 à $N - 1$. Ces calculs peuvent être fait par n'importe quelle méthode précédente et les résultats conservés dans une table.
- Réduction : trouver $y \in [1, 10[$ et $k \in \mathbb{Z}$ tel que $x = y \cdot 10^k$. Utiliser la fonction `reduction_intervalle_10()`.
- Poser $p = \ln(10)$.
- Pour i allant de 0 à $N - 1$:
 - Soit $q = 1 + 10^{-i}$.
 - Tant que $qy \leq 10$, faire :
 - $y \leftarrow qy$
 - $p \leftarrow p - \ln(q)$
- Renvoyer $p + k \ln(10)$ comme approximation de $\ln(x)$.

Commentaires. Nous n'expliquons pas pourquoi cet algorithme fonctionne mais voici pourquoi il est performant : cet algorithme ne fait aucune multiplication, mais seulement des additions, des décalages de virgules et des consultations dans une table pré-établie. En effet, à chaque étape, il y a la multiplication $q \times y$ à calculer, mais c'est une « fausse » multiplication :

$$q \cdot y = (1 + 10^{-i}) \times y = y + \frac{y}{10^i}$$

Or diviser un nombre par une puissance de 10 revient simplement à décaler la virgule à droite. Par exemple :

$$(1 + 10^{-2}) \times 8.765 = 8.765 + \frac{8.765}{100} = 8.765 + 0.08765 = 8.85265$$

On n'a effectué que des additions et des décalages de virgules.

4. Algorithme de Briggs.

L'algorithme suivant a permis à Briggs en 1624 de calculer à la main le logarithme de 30 000 nombres avec 14 décimales après la virgule.

L'idée est basée sur la propriété :

$$\ln(\sqrt{x}) = \frac{1}{2} \ln(x).$$

Autrement dit $\ln(x^{1/2}) = \frac{1}{2} \ln(x)$, puis d'itérer le processus : $\ln(\sqrt{\sqrt{x}}) = \frac{1}{2} \ln(\sqrt{x})$, autrement dit $\ln(x^{1/4}) = \frac{1}{4} \ln(x)$. Puis par récurrence, on calcule $\ln(x^{1/2^n}) = \frac{1}{2^n} \ln(x)$. Au bout d'un certain nombre de racines carrées successives ($n = 54$ pour Briggs !) on obtient

$$x^{\frac{1}{2^n}} \simeq 1$$

On utilise alors que $\ln(u) \simeq u - 1$ si u est suffisamment proche de 1 (autrement dit $\ln(1 + v) \simeq v$ si v est proche de 0).

Exemple. On souhaite calculer une valeur approchée de $\ln(3)$.

- $n = 0$, $x = 3$,
- $n = 1$, $x^{1/2} = \sqrt{x} = \sqrt{3} = 1.7320\dots$
- $n = 2$, $x^{1/4} = \sqrt{\sqrt{x}} = \sqrt{\sqrt{2}} = \sqrt{1.7320\dots} = 1.3160\dots$
- $n = 3$, $x^{1/8} = \sqrt{\sqrt{\sqrt{x}}} = \sqrt{x^{1/4}} = \sqrt{1.3160\dots} = 1.1472\dots$
- $n = 4$, $x^{1/16} = 1.0710\dots$
- $n = 5$, $x^{1/32} = 1.0349\dots$

Quand on considère que l'on est suffisamment proche de 1 on utilise l'approximation :

$$\ln(1 + 0.0349\dots) \simeq 0.0349$$

On a donc $\ln(3^{1/32}) \simeq 0.0349$ et donc $\frac{1}{32} \ln(3) \simeq 0.0349$ ce qui donne

$$\ln(3) \simeq 32 \times 0.0349 \simeq 1.1168$$

C'est une approximation à 0.02 près de $\ln(3) = 1.0986\dots$

Programme l'algorithme suivant en une fonction `logarithme_briggs(x, epsilon)` renvoyant le logarithme de x , selon un certain paramètre de précision ϵ . Pour $x = 1.543$, et $\epsilon = 10^{-10}$ quelle approximation de $\ln(x)$ obtiens-tu ? Combien a-t-il fallu extraire de racines carrées ? Compare avec les fonctions précédentes.

Algorithme.

- Entrée : un nombre $x > 0$, une précision ϵ .
- Sortie : une approximation de $\ln(x)$.

Descente.

- Poser $n = 0$.
- Tant que $|x - 1| > \epsilon$, faire :
 - $x \leftarrow \sqrt{x}$
 - $n \leftarrow n + 1$

Remontée.

- Poser $\ell = x - 1$.
- Pour i allant de 0 à $n - 1$, faire :
 - $\ell \leftarrow 2\ell$
- Renvoyer ℓ comme approximation de $\ln(x)$.

Intégrale

Nous allons étudier différentes techniques pour calculer des valeurs approchées d'intégrales.

Cours 1 (Primitive).

Soit $f : [a, b] \rightarrow \mathbb{R}$. Une **primitive** de f est une fonction dérivable $F : [a, b] \rightarrow \mathbb{R}$ tel que $F'(x) = f(x)$ pour tout $x \in [a, b]$. Si on sait calculer une primitive alors on sait calculer l'intégrale de f :

$$\int_a^b f(t) dt = F(b) - F(a).$$

Exemple : soit $f(x) = x^2$, une primitive de la fonction f est la fonction F définie par $F(x) = \frac{1}{3}x^3$, donc par exemple $\int_0^1 t^2 dt = F(1) - F(0) = \frac{1}{3}$.

Activité 1 (Primitive).

Objectifs : vérifier expérimentalement si une fonction donnée F est une primitive de f .

1. Programme une fonction `verification_primitive(f, F, a, b, n, epsilon)` qui vérifie expérimentalement que F est bien une primitive de f sur l'intervalle $[a, b]$ (n est un entier donné, par exemple $n = 10$ et ϵ une marge d'erreur, par exemple $\epsilon = 0.001$).

Méthode. Vérifie que pour $n + 1$ valeurs x de $[a, b]$ on a $F'(x) \simeq f(x)$. Dans le détail :

- soit $x_k = a + k \frac{b-a}{n}$, $k = 0, 1, \dots, n$;
- on calcule une valeur approchée de $F'(x_k)$ en utilisant la fonction `derivee()` du chapitre « Dérivée »;
- on vérifie $F'(x_k) \simeq f(x_k)$ en testant si $|F'(x_k) - f(x_k)| \leq \epsilon$ pour tout $k = 0, \dots, n$.

2. Écris une fonction `integrale_primitive(F, a, b)` qui calcule l'intégrale d'une fonction f connaissant une primitive F .

3. Application.

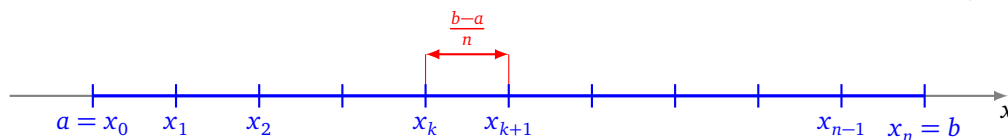
- (a) Calcule l'aire sous la parabole d'équation $y = x^2$ entre les droites verticales d'équation $(x = 1)$ et $(x = 2)$ et au-dessus de l'axe des abscisses.
- (b) Même question avec l'aire sous le graphe de $f(x) = \sin(x)$ sur $[0, \pi]$.

Cours 2 (Calcul approché d'une intégrale).

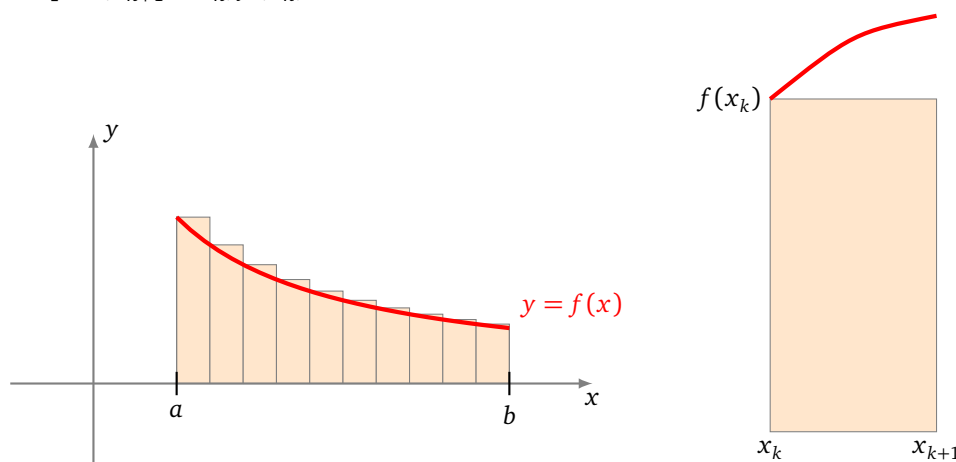
Il n'est pas toujours possible de calculer une primitive pour une fonction $f : [a, b] \rightarrow \mathbb{R}$. On va déterminer des valeurs approchées de $\int_a^b f(t) dt$.

Les trois méthodes d'approximation que l'on va étudier sont toutes basées sur le même principe :

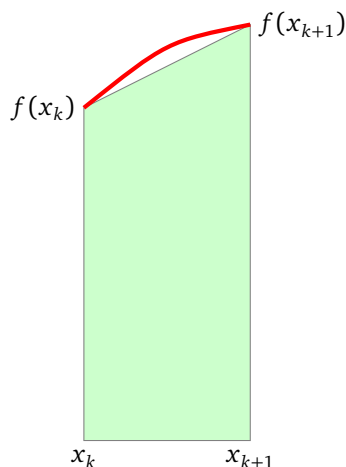
- On divise l'intervalle $[a, b]$ en n sous-intervalles en posant $x_k = a + k \frac{b-a}{n}$ pour $0 \leq k \leq n$. Alors $x_0 = a$ et $x_n = b$ et chaque sous-intervalle $[x_k, x_{k+1}]$ est de longueur constante $x_{k+1} - x_k = \frac{b-a}{n}$.



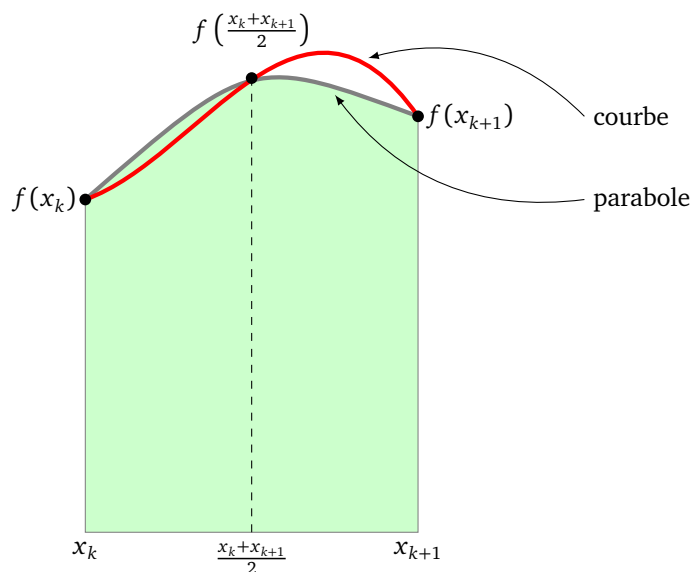
- Sur chaque sous-intervalle $[x_k, x_{k+1}]$, on approche l'aire sous la courbe par l'aire d'une figure géométrique simple.
- Méthode des rectangles.** La *méthode des rectangles (à gauche)* consiste à approcher l'aire sous la courbe par l'aire de rectangles. La hauteur de chaque rectangle est la valeur à gauche de f sur le sous-intervalle (voir les figures ci-dessous). Pour un intervalle élémentaire, cela revient à approcher $\int_{x_k}^{x_{k+1}} f(t) dt$ par $(x_{k+1} - x_k)f(x_k)$.



- Méthode des trapèzes.** On approche l'aire sous la courbe d'un intervalle élémentaire, par l'aire d'un trapèze.



- Méthode de Simpson.** On approche la courbe sur chaque intervalle élémentaire par une branche de parabole.



Activité 2 (Calcul approché d'intégrales).

Objectifs : programmer la méthode des rectangles, des trapèzes et de Simpson.

$$I = \int_a^b f(t) dt$$

1. Méthode des rectangles (à gauche).

Écris une fonction `integrale_rectangles(f, a, b, n)` qui renvoie une valeur approchée de l'intégrale I par la formule :

$$S_R(n) = \frac{b-a}{n} \sum_{k=0}^{n-1} f(x_k)$$

Application. Calcule une valeur approchée de $I_1 = \int_1^2 \frac{1}{t} dt$. Compare avec la valeur exacte (obtenue par primitive). À partir de quelle valeur de n obtiens-tu 3 chiffres exacts après la virgule ? Et pour obtenir 10 chiffres exacts ?

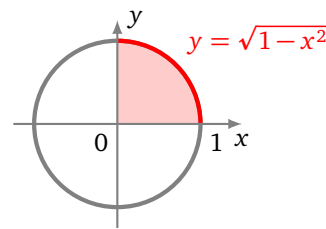
2. Méthode des trapèzes.

Écris une fonction `integrale_trapezes(f, a, b, n)` qui renvoie une valeur approchée de l'intégrale I par la formule :

$$S_T(n) = \frac{b-a}{n} \sum_{k=0}^{n-1} \frac{f(x_k) + f(x_{k+1})}{2}$$

- *Application.* Recommence le calcul de $I_1 = \int_1^2 \frac{1}{t} dt$. Cherche quelles valeurs de n permettent d'avoir 3 chiffres exacts après la virgule, puis 10 chiffres.

- *Application.* Fais le même travail pour calculer une valeur approchée de l'aire d'un disque de rayon 1 par la formule $I_2 = 4 \int_0^1 \sqrt{1-t^2} dt$.



3. Méthode de Simpson.

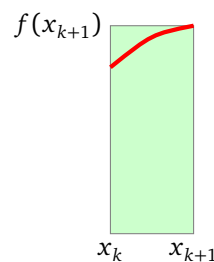
Écris une fonction `integrale_simpson(f, a, b, n)` qui renvoie une valeur approchée de l'intégrale I par la formule :

$$S_S(n) = \frac{b-a}{n} \sum_{k=0}^{n-1} \frac{f(x_k) + 4f\left(\frac{x_k+x_{k+1}}{2}\right) + f(x_{k+1})}{6}$$

- *Application.* Recommence le calcul de $I_1 = \int_1^2 \frac{1}{t} dt$ et trouve les valeurs de n qui permettent d'avoir 3 chiffres exacts après la virgule, puis 10 chiffres.
- *Application.* Fais le même travail pour $I_2 = 4 \int_0^1 \sqrt{1-t^2} dt$.
- *Application.* Trouve une valeur approchée de $I_3 = 4 \int_0^1 \frac{1}{1+t^2} dt$.

4. Bonus.

- Programme une fonction qui calcule une valeur approchée d'une intégrale par la méthode des rectangles à droite : c'est-à-dire que sur l'intervalle $[x_k, x_{k+1}]$ on approche l'intégrale par le rectangle de hauteur $f(x_{k+1})$ (et pas celui de hauteur $f(x_k)$).
- Montre que dans le cas d'une fonction monotone (croissante ou bien décroissante) les deux méthodes des rectangles (droite et gauche) fournissent un encadrement de l'intégrale. Dédus-en des encadrement des intégrales I_1, I_2, I_3 .
- Pour la méthode des trapèzes, essaie d'écrire ta fonction de sorte qu'elle ne calcule qu'une seule fois chaque $f(x_k)$.
- Projet.* Réalise la visualisation graphique des différentes méthodes.

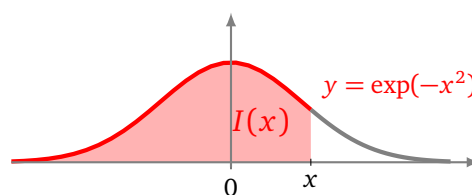
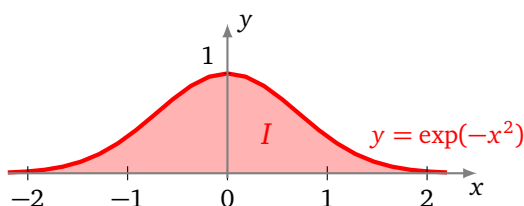


Activité 3 (Intégrale de Gauss).

Objectifs : calculer une valeur approchée de l'intégrale de Gauss.

On note

$$I = \int_{-\infty}^{+\infty} e^{-t^2} dt \quad \text{et} \quad I(x) = \int_{-\infty}^x e^{-t^2} dt.$$



Dans les exemples précédents, les intégrales à calculer avaient une valeur bien connue. C'est aussi le cas de l'intégrale de Gauss I qui vaut $I = \sqrt{\pi}$. Par contre, en général, on ne saura pas calculer la valeur exacte d'une intégrale : c'est le cas des intégrales $I(x)$. D'où l'intérêt d'en trouver des valeurs approchées.

1. Programme une fonction `integrale_gauss1()` qui renvoie une valeur approchée de I et compare ta valeur avec $\sqrt{\pi}$.

Indication. Définis une grande valeur pour l'infini, par exemple en posant $N = 25$. Au lieu de calculer $\int_{-\infty}^{+\infty} f(t) dt$, tu calcules $\int_{-N}^{+N} f(t) dt$ (pour $|x| \geq 25$, $\exp(-x^2)$ est presque nul).

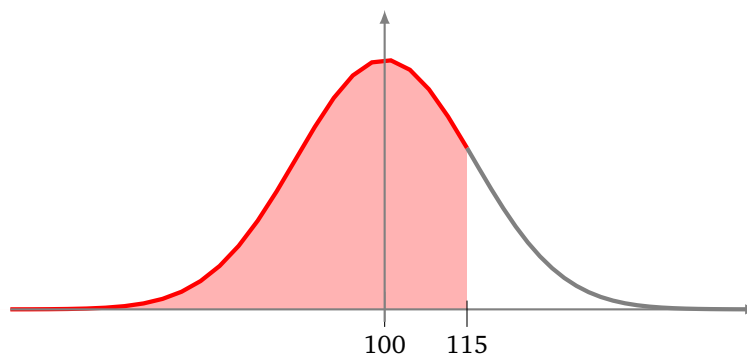
2. Programme une fonction `integrale_gauss2(x)` qui renvoie une valeur approchée de $I(x)$.
3. Pour les calculs de probabilités, nous aurons besoin de calculer la loi normale dont la fonction de répartition est donnée par

$$I_{\mu, \sigma^2}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^x e^{-\frac{(t-\mu)^2}{\sigma^2}} dt$$

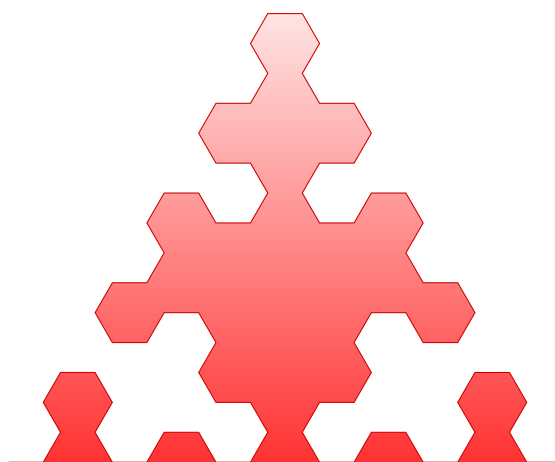
où μ est l'espérance et σ^2 la variance (σ est l'écart-type).

Programme une fonction `integrale_gauss3(x, mu, sigma2)` qui renvoie $I_{\mu, \sigma^2}(x)$.

4. On modélise la répartition des personnes selon leur QI par une courbe de Gauss de paramètre $\mu = 100$ (le QI moyen) et $\sigma^2 = 225$ (écart-type $\sigma = 15$). Avec ces paramètres $I_{\mu, \sigma^2}(x)$ représente le pourcentage de personnes ayant un QI inférieur à x . Par exemple $I(100) = 0.5$, donc 50% de la population a un QI inférieur à 100. Quel pourcentage de la population a un QI supérieur à 115 ?



DEUXIÈME PARTIE



INFORMATIQUE AVEC MATHÉMATIQUES

Programmation objet

Avec Python tout est objet : un entier, une chaîne, une liste, une fonction. . . Nous allons voir comment définir nos propres objets.

Cours 1 (Programmation objet : la classe !).

Un **objet** est une entité qui regroupe à la fois des variables et des fonctions. Le premier intérêt est qu'un objet est indépendant et auto-suffisant puisqu'il contient tout ce qu'il faut pour être utilisé, il permet d'éviter le recours aux variables globales par exemple.

Un objet est défini comme une **instance** d'une **classe**, c'est-à-dire un élément d'une catégorie. Voici un exemple de la vie courante : on considère la classe *Chien*, alors mon chien *Médor* est un objet, appartenant à la classe *Chien*. Note que *Chien* est un concept, mais que *Médor* est bien réel. Mon autre chien *Foulcan* est aussi une instance de *Chien*.

Voici comment définir le début d'une classe `Vecteur()` afin de modéliser des vecteurs de l'espace :

```
class Vecteur:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

Pour l'instant un vecteur est un concept auquel sont rattachés trois nombres (x, y, z) . Le mot `self` fait référence à l'objet lui-même mais dont on ne connaît pas encore le nom (ce sera `V` ou bien `V1`, `V2`...).

Et voici un objet défini à partir de cette classe :

```
V = Vecteur(1, 2, 3)
```

Cet objet possède trois **attributs** :

```
V.x      V.y      V.z
```

qui valent ici respectivement 1, 2 et 3. Autre exemple, le calcul `V.x + V.y + V.z` renvoie 6. Je peux changer une de ces valeurs comme pour une variable classique (même si ce n'est pas la manière recommandée), par exemple :

```
V.x = 7
```

Maintenant `V.x + V.y + V.z` vaut 12.

Tu peux définir plusieurs objets qui seront indépendants les uns des autres :

```
V1 = Vecteur(1, 2, 3)    V2 = Vecteur(1, 0, 0)
```

Ainsi par exemple `V1.y` vaut 2, `V2.y` vaut 0.

Cours 2 (Programmation objet : de la méthode.).

On a vu comment attribuer des variables à un objet. Nous allons voir comment lui associer des fonctions.

Pour un objet, une fonction associée s'appelle une *méthode*.

Si on reprend l'exemple de la classe *Chien*, on pourrait lui associer une méthode *Viens_ici_!*. On peut donc demander *Médor.Viens_ici_!* ou bien *Foulcan.Viens_ici_!* pour appeler chacun de nos chiens.

Complétons notre classe *Vecteur()* pour lui associer trois nouvelles méthodes :

```
class Vecteur:
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z

    def norme(self):
        N = sqrt(self.x**2 + self.y**2 + self.z**2)
        return N

    def produit_par_scalaire(self,k):
        W = Vecteur(k*self.x,k*self.y,k*self.z)
        return W

    def addition(self,other):
        W = Vecteur(self.x+other.x,self.y+other.y,self.z+other.z)
        return W
```

- La méthode *norme()* renvoie la norme $\sqrt{x^2 + y^2 + z^2}$ d'un vecteur (x, y, z) (il faut importer le module *math*). Par exemple pour *V = Vecteur(1,2,3)*, on demande sa norme par la commande :

V.norme()

qui renvoie ici une valeur approchée de $\sqrt{14} = 3.74\dots$

La méthode *norme()* est définie comme une fonction classique, le paramètre prend le nom de *self* et correspond à l'objet (le vecteur *V* pour notre exemple). On récupère les coordonnées par *self.x*, *self.y*, *self.z* (pour notre exemple cela correspond à *V.x*, *V.y*, *V.z*).

- La méthode *produit_par_scalaire(self,k)* multiplie les coordonnées d'un vecteur par un réel *k*. Par exemple pour *V = Vecteur(1,2,3)* alors la commande :

W = V.produit_par_scalaire(7)

définit un nouvel objet *Vecteur()*, noté *W*, représentant le vecteur $\vec{w} = (7, 14, 21)$. Maintenant *W* est un objet de classe *Vecteur()* comme les autres et on peut par exemple calculer sa norme par *W.norme()*. La méthode *produit_par_scalaire()* est définie à l'aide de deux paramètres. Le premier est obligatoirement *self* et fait toujours référence à l'objet traité. Le second est ici le facteur *k*. Lorsque l'on appelle la méthode cela devrait être *produit_par_scalaire(V,7)* mais la syntaxe des objets est *V.produit_par_scalaire(7)* (le premier argument passe devant le nom de la méthode, les autres arguments sont décalés).

- La méthode *addition(self,other)* renvoie le vecteur somme de deux vecteurs, cela correspond à l'opération

$$(x, y, z) + (x', y', z') = (x + x', y + y', z + z')$$

Voici un exemple d'utilisation :

```
V1 = Vecteur(1,2,3)
V2 = Vecteur(1,0,-4)
V3 = V1.addition(V2)
```

On définit deux vecteurs $\vec{v}_1$ et $\vec{v}_2$, leur somme $\vec{v}_3$ vaut ici $(2, 2, -1)$.

La méthode `addition()` est définie à l'aide de deux paramètres : le premier est toujours `self` et le second se nomme ici `other` pour signifier qu'il s'applique à un autre objet de la même classe. Pour notre exemple le paramètre `self` correspond à l'argument `V1` et le paramètre `other` à l'argument `V2`.

Cours 3 (Programmation objet : convivialité).

Complétons notre classe `Vecteur()` afin de permettre un joli affichage et d'additionner les vecteurs à l'aide de l'opérateur « + ».

```
class Vecteur:
    def __init__(self,x,y,z):
        ...

    def __str__(self):
        ligne = "("+str(self.x)+"," +str(self.y)+"," +str(self.z)+")"
        return ligne

    def __add__(self,other):
        W = Vecteur(self.x+other.x,self.y+other.y,self.z+other.z)
        return W
```

- La méthode `__str__()` (le nom est réservé) renvoie ici un bel affichage du vecteur. Par exemple avec `V = Vecteur(1,2,3)` alors :

```
print(V.__str__())    affiche    (1,2,3).
```

Mais ce n'est pas comme cela qu'on l'utilise car une fois que la méthode `__str__()` est définie alors la commande :

```
print(V)    affiche aussi    (1,2,3).
```

C'est très pratique !

- La méthode `__add__()` a exactement la même définition que la méthode `addition()` définie précédemment. Avec `V1 = Vecteur(1,2,3)` et `V2 = Vecteur(1,0,-4)` on pourrait l'utiliser par :

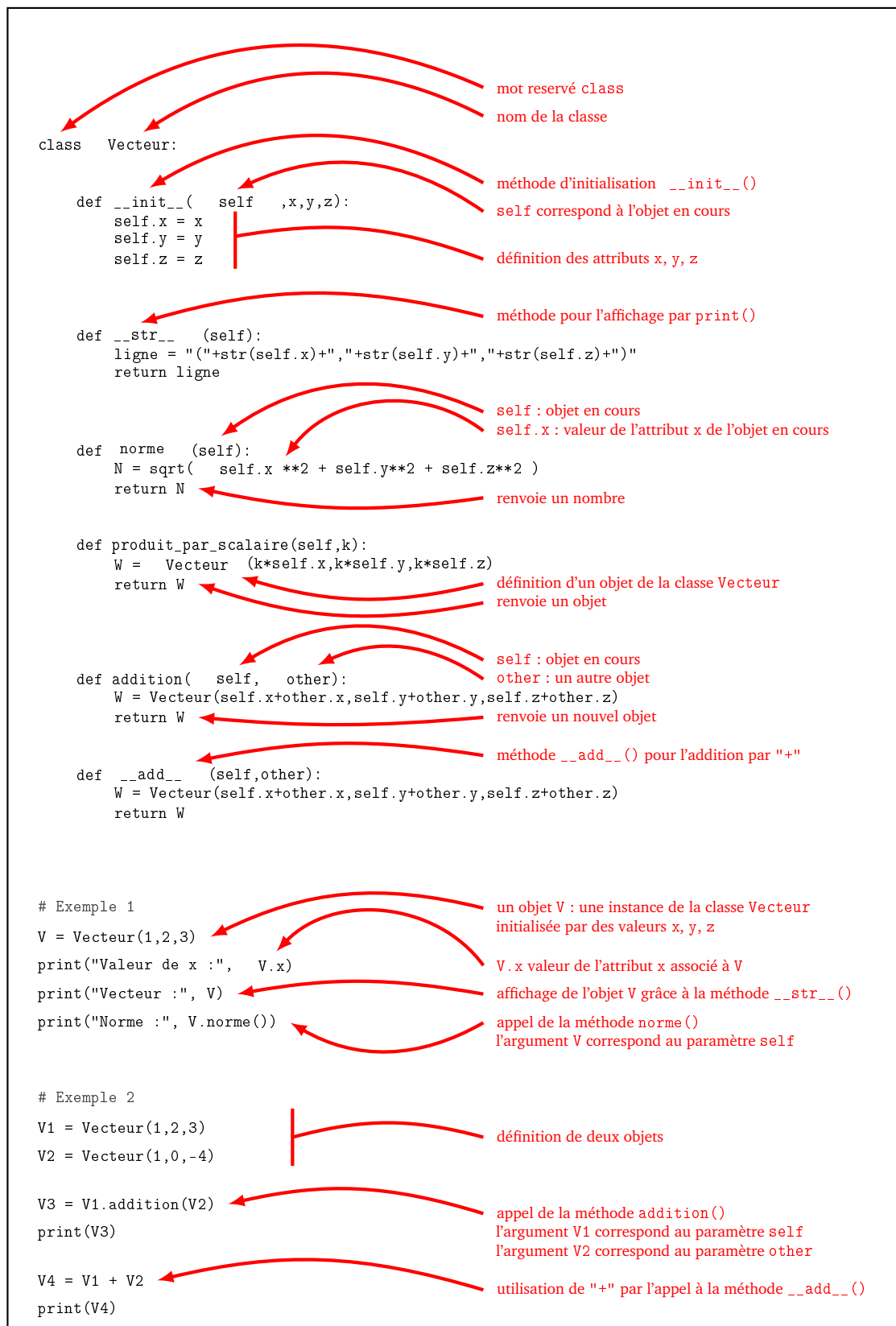
```
V3 = V1.__add__(V2)
```

Mais comme on a utilisé le nom réservé `__add__()` alors cela a défini l'opérateur « + » et il est beaucoup plus agréable d'écrire simplement :

```
V3 = V1 + V2
```

Cours 4 (Programmation objet : résumé.).

Voici la définition complète de la classe `Vecteur()` accompagnée d'un résumé des explications.



Cours 5 (Matrice 2×2).

- Une matrice 2×2 est un tableau :

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

- On peut additionner deux matrices et multiplier une matrice par un réel :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} a+a' & b+b' \\ c+c' & d+d' \end{pmatrix} \quad k \cdot \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} ka & kb \\ kc & kd \end{pmatrix}$$

- Le produit de deux matrices est défini par la formule suivante :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} aa' + bc' & ab' + bd' \\ ca' + dc' & cb' + dd' \end{pmatrix}$$

- La trace et le déterminant sont deux réels associés à une matrice :

$$\text{tr} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = a + d \quad \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

- Si une matrice $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ a son déterminant non nul alors elle admet un inverse :

$$M^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

de sorte que

$$M \times M^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Activité 1 (Matrices).

Objectifs : définir des matrices comme des objets.

On commence à définir une classe `Matrice()` pour stocker des matrices 2×2 et leurs opérations.

```
class Matrice:
    def __init__(self, a, b, c, d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d
```

Une matrice M sera donc définie par la commande :

```
M = Matrice(1, 2, 3, 4)
```

pour définir la matrice

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

1. Définis une méthode `__str__(self)` qui permet l'affichage de la matrice. Cette méthode permet aussi d'obtenir l'affichage à l'aide `print()`. Si on a défini `M = Matrice(1, 2, 3, 4)` alors la commande `print(M)` équivaut à la commande `print(M.__str__())` et affiche à l'écran :

```
1  2
3  4
```

2. Définis une méthode `trace(self)` et une méthode `determinant(self)` qui calcule la trace et le déterminant d'une matrice. Pour notre exemple `M.trace()` renvoie 5 et `M.determinant()` renvoie -2.
3. Définis une méthode `produit_par_scalaire(self, k)` qui renvoie la matrice correspondant au produit de chaque coefficient par le réel k . Ainsi, à partir de notre matrice M , on peut définir une

nouvelle matrice M' par la commande `MM = M.produit_par_scalaire(5)` qui correspond à $M' = \begin{pmatrix} 5 & 10 \\ 15 & 20 \end{pmatrix}$.

4. Définis une méthode `inverse(self)` qui calcule l'inverse d'une matrice (et renvoie `None` si le déterminant est nul). Pour notre exemple `M.inverse()` renvoie la matrice qui correspond à

$$M^{-1} = \begin{pmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{pmatrix}$$

5. (a) Définis une méthode `addition(self, other)` qui calcule la somme de deux matrices. Par exemple avec :

$$M1 = \text{Matrice}(4, 3, 2, 1) \quad M2 = \text{Matrice}(1, 0, -1, 1)$$

puis :

$$M3 = M1.addition(M2)$$

alors $M3$ correspond à la matrice :

$$M_3 = M_1 + M_2 = \begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} 5 & 3 \\ 1 & 2 \end{pmatrix}$$

- (b) C'est beaucoup mieux de nommer cette méthode `__add__(self, other)` puisque cela permet d'écrire tout simplement :

$$M3 = M1 + M2$$

6. (a) Définis une méthode `multiplication(self, other)` qui calcule le produit de deux matrices. Par exemple avec nos matrices M_1 et M_2 :

$$M4 = M1.multiplication(M2)$$

correspond à la matrice :

$$M_4 = M_1 \times M_2 = \begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 1 & 1 \end{pmatrix}.$$

- (b) C'est beaucoup mieux de nommer cette méthode `__mul__(self, other)` puisque cela permet d'écrire tout simplement :

$$M4 = M1 * M2$$

Vérifie que $M_1 \times M_2$ et $M_2 \times M_1$ ne sont **pas** les mêmes matrices !

- (c) Vérifie sur plusieurs exemples qu'une matrice, multipliée par son inverse, vaut la matrice identité $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Par exemple on a bien `M1 * M1.inverse()` qui vaut la matrice identité.

7. *Application.* La suite de Fibonacci est définie par récurrence :

$$F_0 = 1 \quad F_1 = 1 \quad F_{n+2} = F_{n+1} + F_n \quad \text{pour } n \geq 0.$$

Chaque terme est donc la somme des deux termes précédents. Les premiers termes sont :

$$F_0 = 1 \quad F_1 = 1 \quad F_2 = 2 \quad F_3 = 3 \quad F_4 = 5 \quad F_5 = 8 \quad F_6 = 13 \dots$$

Une autre façon de calculer F_n est d'utiliser des matrices. Soit

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Alors

$$M^n = \underbrace{M \times M \times \dots \times M}_{n \text{ fois}} = \begin{pmatrix} F_{n-2} & F_{n-1} \\ F_{n-1} & F_n \end{pmatrix}$$

Autrement dit F_n est le dernier coefficient de la matrice M^n .

Calcule F_{100} à l'aide des matrices.

I

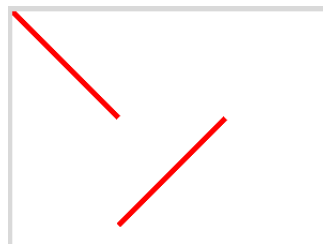
Activité 2 (Tortue basique).

Objectifs : programmer une tortue basique (sur le principe de Scratch) qui réagit à des instructions simples.

Voici le début de la définition d'une classe `TortueBasique()` qui définit quatre attributs : les coordonnées x et y de la position courante de la tortue (située au départ en $(0,0)$), la position du stylo (`trace` vaut « Vrai » ou « Faux »), la couleur du stylo :

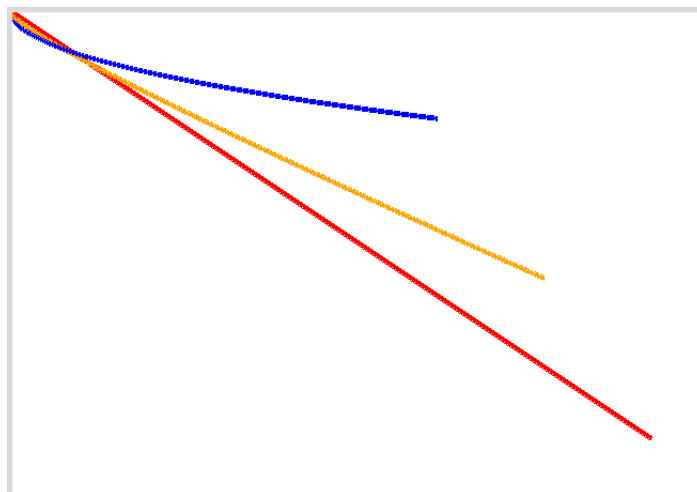
```
class TortueBasique:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.trace = True
        self.couleur = 'red'
```

1. Copie puis complète cette définition avec une méthode `renvoyer_xy(self)` qui renvoie les coordonnées (x, y) de la position courante.
2. Complète avec une méthode `aller_a_xy(self, x, y)` qui déplace la tortue à la position (x, y) indiquée. Si `trace` vaut « Vrai », trace un segment entre l'ancienne et la nouvelle position.
Indication. Pour le tracé utilise le module `tkinter` (voir plus bas).
3. Complète avec des méthodes `abaisser_stylo(self)`, `relever_stylo(self)` qui changent la valeur de l'attribut `trace` et une méthode `changer_couleur(self, couleur)` puis dessine la figure suivante :



4. Définis une `tortue1` (rouge), une `tortue2` (bleue). Définis une `tortue3` (orange) qui à chaque déplacement de `tortue1` et `tortue2` se place au milieu de ces deux tortues.

Sur le dessin ci-dessous : `tortue1` se déplace en $(\frac{3}{2}i, i)$ (pour i allant de 0 à 400) ; `tortue2` se déplace successivement en $(i, 5\sqrt{i})$. Pour chaque i , on récupère les positions de ces deux tortues et `tortue3` se place au milieu.



Voici un exemple d'utilisation de notre tortue en utilisant le module `tkinter` pour l'affichage (voir le dessin ci-dessus).

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(side=LEFT, padx=5, pady=5)

tortue = TortueBasique()

tortue.aller_a_xy(100,100)
tortue.relever_stylo()
tortue.aller_a_xy(200,100)
tortue.abaisser_stylo()
tortue.aller_a_xy(100,200)

root.mainloop()
```

Cours 6 (Programmation objet : héritage).

La programmation objet possède un autre intérêt : à partir d'une classe d'objets on peut en définir d'autres, en récupérant certaines des fonctionnalités et en ajoutant de nouvelles. C'est très utile par exemple pour reprendre et compléter du code écrit par d'autres. C'est la notion d'*héritage*.

Voyons un exemple : on veut créer un jeu vidéo où il faut combattre des ennemis. Il y a différents types d'ennemis mais ils ont tous des caractéristiques communes : une position (x, y) et des points de vie. Voici une classe `Ennemi()` avec une méthode qui affiche les points de vie restant et une autre qui diminue les points de vie après avoir été attaqué.

```
class Ennemi():
    def __init__(self, x, y, vie):
        self.x = x
        self.y = y
        self.vie = vie

    def affiche_vie(self):
        print("Vie =", self.vie)

    def perd_vie(self, n):
        self.vie = self.vie - n
```

Voici deux objets immobiles (des tours) définis par cette classe `Ennemi()` et quelques actions.

```
tour = Ennemi(1, 2, 100)
super_tour = Ennemi(5, 3, 200)
tour.affiche_vie()
tour.perd_vie(50)
tour.affiche_vie()
```

Pour les ennemis passifs (qui peuvent être attaqués, mais ne peuvent pas attaquer) la classe `Ennemi()` est bien adaptée. Par contre cette classe n'est pas assez évoluée pour des ennemis plus performants.

Prenons l'exemple d'un zombie : en plus des caractéristiques déjà décrites, il est actif (il peut vous attaquer, se déplacer...). Une solution est de programmer une classe `Zombie()` depuis zéro, mais ce serait dommage car une partie du travail a été faite avec la classe `Ennemi()`. Le plus simple est de récupérer les caractéristiques déjà existantes et d'en ajouter de nouvelles. C'est ce qu'on fait avec la classe `Zombie()` qui hérite des propriétés de la classe `Ennemi()` :

```
class Zombie(Ennemi):
    def __init__(self,x,y,vie,force):
        Ennemi.__init__(self,x,y,vie)
        self.force = force

    def affiche_force(self):
        print("Force =",self.force)
```

Voici un exemple d'utilisation :

```
mechant = Zombie(4,4,100,100)
mechant.affiche_force()
mechant.perd_vie(50)
mechant.affiche_vie()
```

Voici les explications :

- la classe `Zombie()` est définie par l'entête « `class Zombie(Ennemi):` » et ainsi hérite des attributs et des méthodes de la classe `Ennemi()`.
- Une instance de la classe `Zombie()` possède les attributs `x`, `y` et `vie` hérités de `Ennemi()` mais possède en plus des points d'attaques stockés dans `force`.
- La méthode `__init__()` initialise un objet de la classe `Zombie()`. Pour les caractéristiques déjà pré-existantes de la classe mère, on les initialise par `Ennemi.__init__(self,x,y,vie)` et il ne reste plus qu'à initialiser `force`.
- On définit une nouvelle méthode `affiche_force()` qui concerne seulement les `Zombie()`.
- On voit dans l'exemple d'utilisation comment définir un objet de la classe `Zombie()` (avec ses quatre attributs). On peut bien sûr utiliser la méthode `affiche_force()` spécifique à cette classe, mais aussi les méthodes `affiche_vie()` et `perd_vie()` héritées de la classe `Ennemi()`.

Activité 3 (Tortue tournante).

Objectifs : définir une tortue plus performante en se basant sur les propriétés de la tortue basique déjà construite.

On veut améliorer la classe `TortueBasique()` en une classe `TortueTournante()` qui permet de diriger une tortue selon une direction. La classe `TortueTournante()` est donc héritée de la classe `TortueBasique()` et un nouvel attribut `direction` est créé. Le début de la définition est donc :

```
class TortueTournante(TortueBasique):
    def __init__(self):
        TortueBasique.__init__(self)
        self.direction = 0
```

L'attribut `direction` correspond à l'angle en degrés vers lequel pointe la tortue. L'angle 0 correspond à la droite, l'angle 90 degrés correspond au Nord.

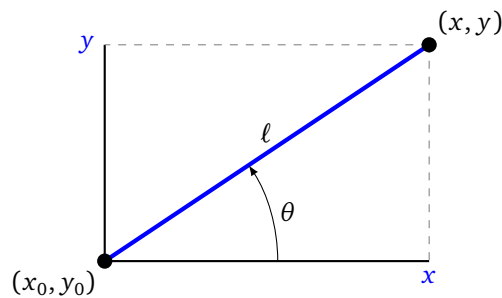
1. Complète la définition de la classe avec une méthode `fixer_direction(self,direction)` qui met à jour la direction courante avec l'angle donné.

Encapsulation. Cette fonction sert juste à éviter d'écrire `tortue.direction = 90` ce qui est déconseillé en dehors de la définition de la classe. Il faut plutôt utiliser `tortue.fixer_direction(90)`. Cette recommandation s'appelle l'**encapsulation**.

2. Complète la définition de la classe avec une méthode `tourner(self, angle)` qui change la direction courante en ajoutant l'angle donné.
3. Complète la définition de la classe avec une méthode `avancer(self, longueur)` qui fait avancer la tortue de la longueur donnée selon sa direction courante.

Voici les formules pour calculer les coordonnées (x, y) du point d'arrivée en fonction du point de départ (x_0, y_0) de la direction θ (en degrés) et de la longueur ℓ :

$$x = x_0 + \ell \cos\left(\frac{2\pi}{360}\theta\right) \quad y = y_0 + \ell \sin\left(\frac{2\pi}{360}\theta\right)$$

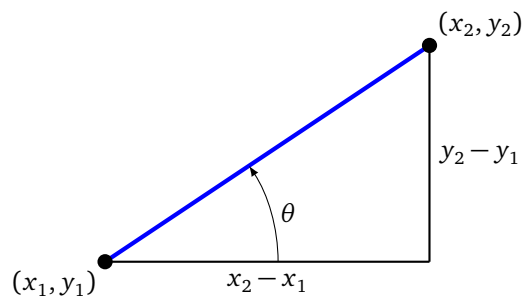


4. Complète la définition de la classe avec une méthode `sorienter_vers(self, other)` qui oriente une tortue `self` en direction de la tortue `other`.

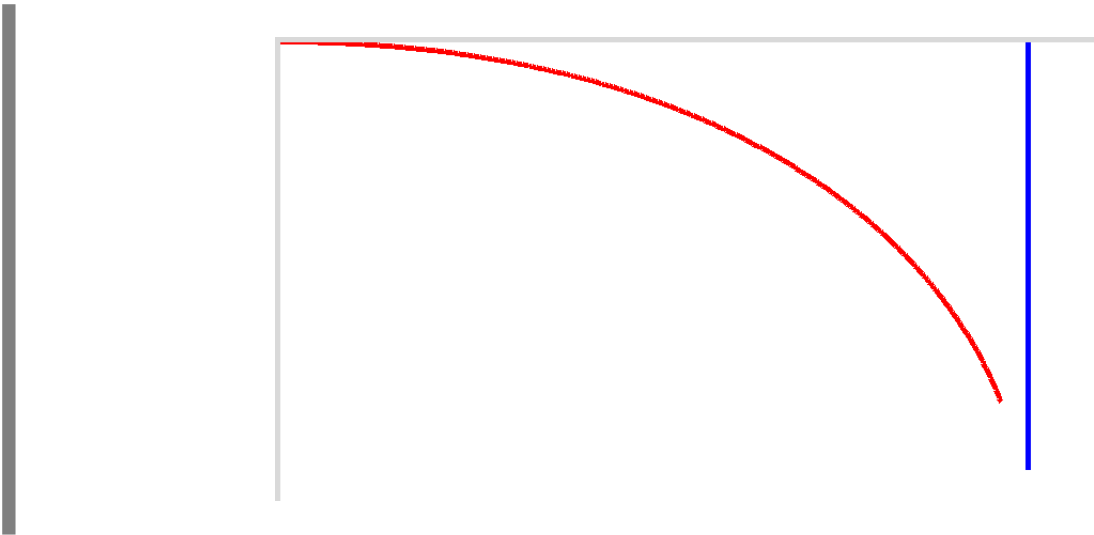
Indications. Si (x_1, y_1) sont les coordonnées d'une première tortue, (x_2, y_2) les coordonnées d'une seconde tortue alors l'angle formé entre l'horizontale et la droite joignant les deux tortues est donné (en degrés) par la formule :

$$\theta = \frac{360}{2\pi} \text{atan2}(y_2 - y_1, x_2 - x_1)$$

où `atan2(y, x)` est une variante de la fonction arctangente disponible dans le module `math`.



Programme une poursuite de tortue : une tortue bleue descend pas à pas, à chaque étape la tortue rouge s'oriente vers la tortue bleue, puis avance. La courbe tracée par la tortue rouge s'appelle une « courbe de poursuite ».



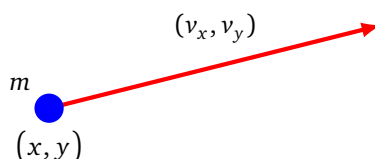
Mouvement de particules

Tu vas simuler le mouvement d'une particule soumise à différentes forces, comme la gravité ou des frottements. Tu appliqueras ceci afin de simuler le mouvement des planètes autour du Soleil. Cette activité utilise la programmation objet.

Cours 1 (Une particule).

Modélisation. Une particule est modélisée par cinq valeurs :

- ses coordonnées x et y ,
- les coordonnées v_x et v_y de sa vitesse,
- sa masse m .



Particule soumise à aucune force. Si aucune force n'agit sur la particule, alors elle conserve sa direction et sa vitesse. Ainsi à l'instant élémentaire suivant, la nouvelle position de la particule est :

$$\begin{cases} x' &= x + v_x \\ y' &= y + v_y \end{cases}$$

La particule a donc un mouvement rectiligne uniforme.

Preuve. La vitesse est la dérivée de la position, ainsi par exemple v_x est la limite du taux d'accroissement $\frac{x(t+dt)-x(t)}{dt}$. En considérant que dt est une durée infinitésimale on obtient ; $x(t + dt) = x(t) + v_x dt$. En choisissant comme unité de temps $dt = 1$, on obtient la formule voulue. Les calculs sont identiques pour v_y .

Particule soumise à une force. Si la particule est soumise à une force $\vec{F}$ dont les composantes sont (F_x, F_y) alors à l'instant élémentaire suivant, la nouvelle **vitesse** de la particule est :

$$\begin{cases} v'_x &= v_x + F_x/m \\ v'_y &= v_y + F_y/m \end{cases}$$

Comme la vitesse est modifiée, cela induit un changement sur la future position.

Preuve. Le principe fondamental de la mécanique affirme que :

$$\vec{F} = m\vec{a}$$

où $\vec{a}$ est le vecteur accélération et m la masse. L'accélération est la dérivée de la vitesse, donc en coordonnées on a :

$$F_x = m \frac{dv_x}{dt} \quad F_y = m \frac{dv_y}{dt}$$

Autrement dit $m \frac{v_x(t+dt) - v_x(t)}{dt} = F_x$. Donc $v_x(t+dt) = v_x(t) + \frac{F_x}{m} dt$. En normalisant à $dt = 1$, on obtient la formule voulue.

S'il y a plusieurs forces $\vec{F}_1, \vec{F}_2 \dots$ alors on les regroupe en un seul vecteur, la force résultante : $\vec{F} = \vec{F}_1 + \vec{F}_2 + \dots$

Activité 1 (Une particule).

Objectifs : programmer le mouvement d'une particule et son affichage.

Informatiquement, on code une particule par un objet de la classe `Particule()` contenant 5 attributs x, y, v_x, v_y, m (correspondant à x, y, v_x, v_y, m). Voici le début de la définition de la classe `Particule()` que tu vas peu à peu compléter :

```
class Particule():
    def __init__(self, x, y, vx, vy, m):
        self.x = x
        self.y = y
        self.vx = vx
        self.vy = vy
        self.m = m

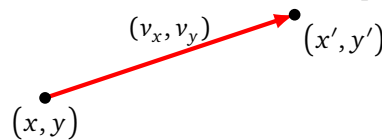
    def __str__(self):
        ligne = "(" + str(self.x) + ", " + str(self.y) + ")",
        (" + str(self.vx) + ", " + str(self.vy) + ")", " + str(self.m)
        return ligne
```

Voici un exemple d'initialisation d'une particule p placée en $(-100, 100)$ avec une vitesse de vecteur $(20, 0)$ (donc horizontale) et de masse $m = 1$. La méthode `__str__()` définie plus haut permet d'afficher proprement l'objet p .

```
p = Particule(-100, 100, 20, 0, 1)
print(p)
```

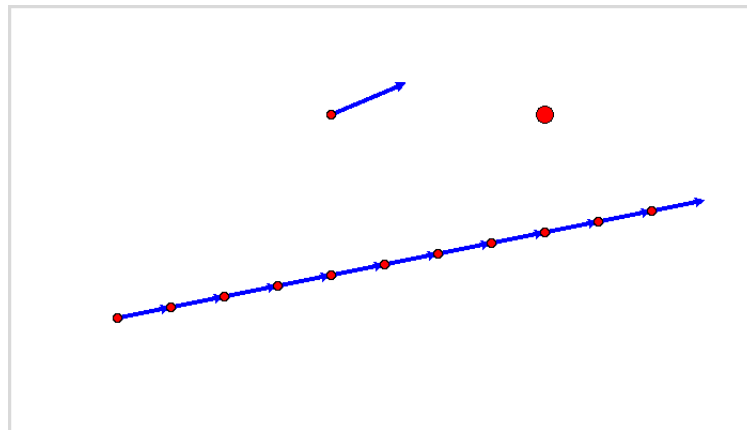
1. Complète la définition de la classe avec une méthode `action_vitesse(self)` qui déplace la particule en suivant le vecteur vitesse. Les nouvelles coordonnées sont données par la formule :

$$\begin{cases} x' &= x + v_x \\ y' &= y + v_y \end{cases}$$



2. Complète la définition de la classe avec une méthode `affiche(self)` qui affiche graphiquement la particule.

Sur la figure ci-dessous en haut à gauche une particule avec son vecteur vitesse, en haut à droite une particule sans affichage de son vecteur vitesse mais de masse plus grosse. En bas une particule qui se déplace suivant son vecteur vitesse (sur 10 unités de temps, en suivant un mouvement rectiligne uniforme).

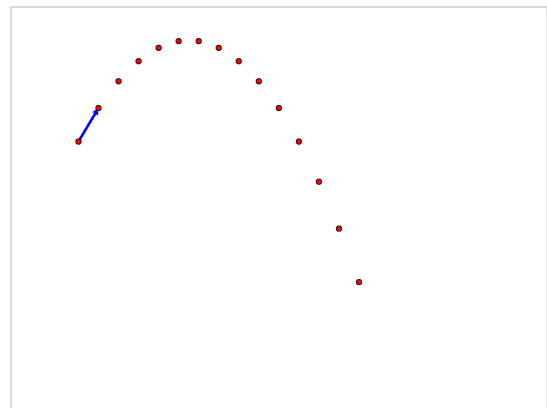
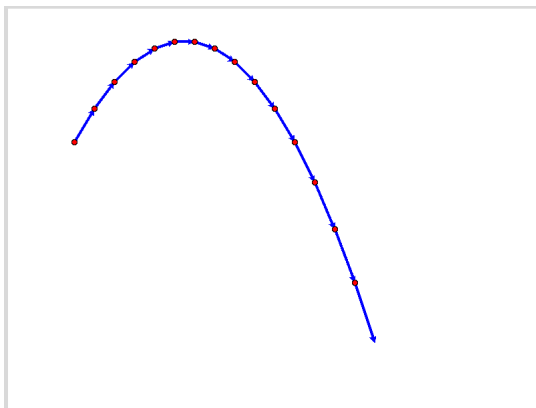


Indications.

- Utilise le module `tkinter` (voir plus bas) (avec une unité qui vaut un pixel, le point (0,0) étant au centre de l'écran). Le passage des coordonnées réelles aux coordonnées graphiques se fait par les formules $i = \text{Largeur}/2 + x$ et $j = \text{Hauteur}/2 - y$.
 - Tu peux définir ta fonction avec une entête `affiche(self, avec_fleche=False)` et laisser le choix de l'affichage du vecteur vitesse sous la forme d'une flèche.
 - Le rayon du disque peut dépendre de la masse.
3. Complète la définition de la classe avec une méthode `action_gravite(self, gravite=0.2)` qui correspond à l'action de la force de gravité sur la particule. Cela correspond à changer la valeur de la vitesse verticale, la nouvelle valeur v'_y étant calculée à partir de l'ancienne valeur v_y par la formule :

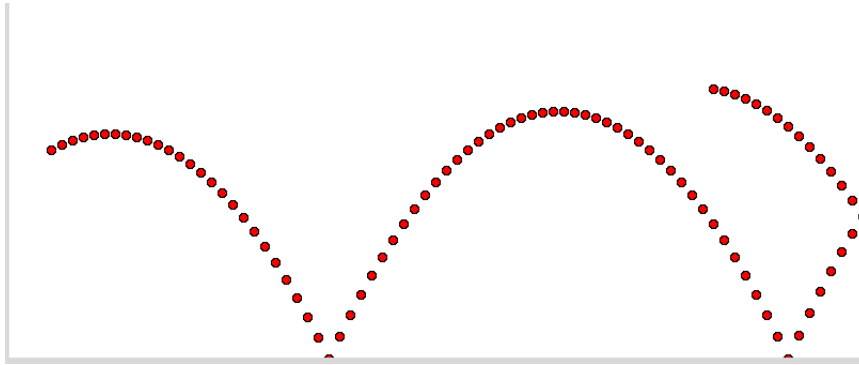
$$v'_y = v_y - g.$$

Teste différentes valeurs de la constante de gravité.



Comme on peut le voir sur les formules (et sur les tracés) il est remarquables que les mouvements de particules de deux masses différentes sont identiques.

4. Complète la définition de la classe avec une méthode `rebondir_si_bord_atteint(self)` qui empêche la particule de sortir de l'écran. Pour cela si x est trop grand ou trop petit alors inverse le signe de v_x (c'est-à-dire $v_x \leftarrow -v_x$). De même pour y .



5. Complète la définition de la classe avec une méthode

`action_frottement(self, frottement=0.005, exposant=2)`

qui correspond à l'action d'une force de frottement.

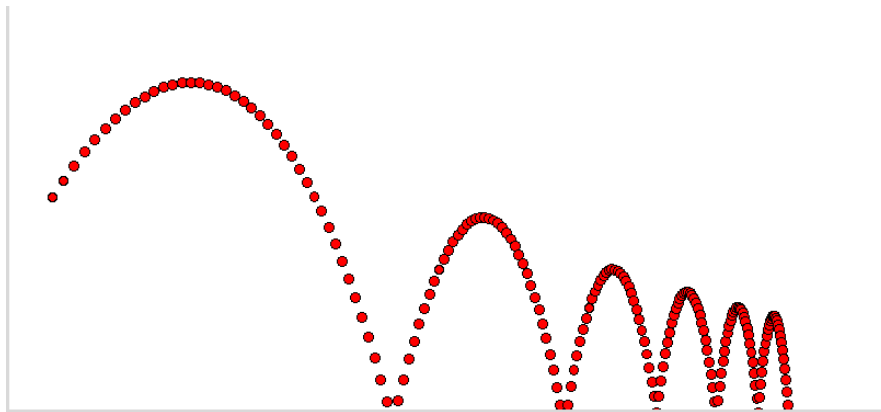
On note

$$v = \sqrt{v_x^2 + v_y^2}.$$

Les nouvelles valeurs du vecteur vitesse sont données par les formules :

$$\begin{cases} v'_x &= v_x - \frac{f}{m} \cdot v^e \cdot \frac{v_x}{v} \\ v'_y &= v_y - \frac{f}{m} \cdot v^e \cdot \frac{v_y}{v} \end{cases}$$

où f est le coefficient de frottement et e l'exposant de frottement. Essaie différentes valeurs du coefficient de frottement et surtout de l'exposant. Un exposant $e = 1$ correspond à un frottement pour une particule se déplaçant à faible vitesse ; un exposant $e = 2$ correspond à une vitesse élevée ; des exposants $1 < e < 2$ sont possibles.



6. Complète la définition de la classe avec une méthode `mouvement(self)` qui regroupe la succession des actions définies : `action_vitesse()`, `action_gravite()`, `action_frottement()`, `rebondir_si_bord_atteint()` Ainsi pour simuler le mouvement d'une particule il suffit d'écrire :

```
# Constantes pour l'affichage
```

```
Largeur = 800
```

```
Hauteur = 600
```

```
# Fenêtre tkinter
```

```
from tkinter import *
```

```
root = Tk()
```

```

canvas=Canvas(root,width=Largeur,height=Hauteur,background="white")
canvas.pack(side=LEFT, padx=5, pady=5)

p = Particule(-300,10,5,2,10)
for k in range(100):
    p.mouvement()
    p.affiche()

root.mainloop()

```

Activité 2 (Particules en mouvement).

Objectifs : afficher une ou plusieurs particules en mouvement.

Voici le « film » de 10 particules, chacune étant lancée à partir du même point mais avec un vecteur vitesse initial différent.



Programme tout cela en utilisant les explications qui suivent.

Voici une classe `TkParticule()` héritée de la classe `Particule()` qui permet d'afficher une particule en mouvement (la particule évolue sans afficher sa trace). Il y a deux attributs supplémentaires : un attribut pour la couleur, un attribut pour l'identifiant `tkinter` de la particule qui permet ensuite à la méthode `affiche()` de déplacer le disque grâce à la méthode `move()` de `tkinter`.

```

class TkParticule(Particule):
    def __init__(self,x,y,vx,vy,m,couleur="red"):
        Particule.__init__(self,x,y,vx,vy,m)
        self.couleur = couleur
        i,j = xy_vers_ij(x,y)
        rayon = min(max(1,m),10)
        # Création de l'objet tkinter
        disque = canvas.create_oval(i-rayon,j-rayon,

```

```

        i+rayon,j+rayon,fill=self.couleur)
    self.id = disque

```

```

def affiche(self):
    canvas.move(self.id,self.vx,-self.vy)

```

La fonction `xy_vers_ij(x,y)` transforme les coordonnées réelles (x,y) en coordonnées graphiques (i,j) .

Voici un lancé de plusieurs particules (on utilise le module `time`).

```

liste_particules=[TkParticule(-300,0,10,j,5,couleur=hasard_couleur())
                  for j in range(10)]
for k in range(200):
    for p in liste_particules:
        p.mouvement()
        p.affiche()

    canvas.update()
    sleep(0.05)

root.mainloop()

```

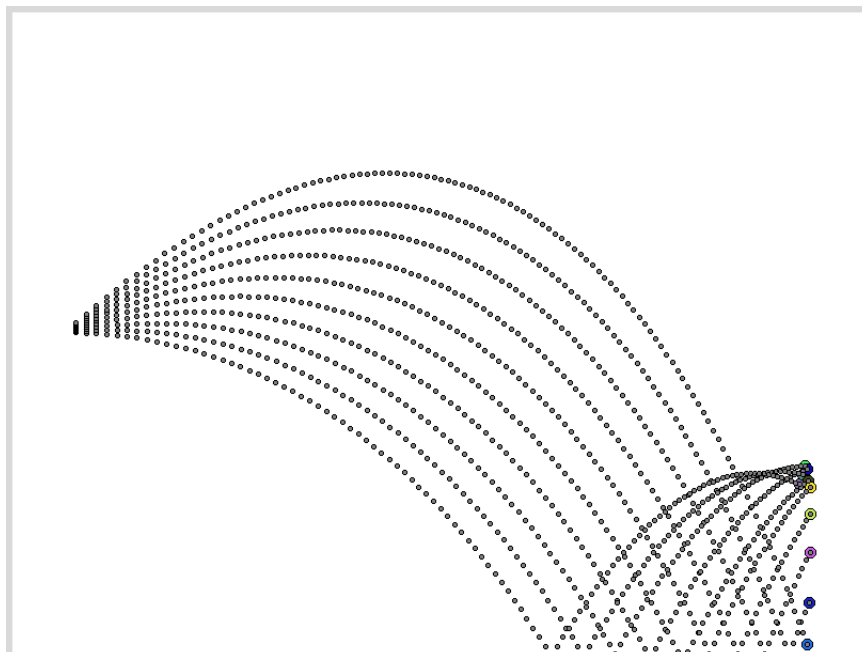
Voici une fonction qui renvoie une couleur au hasard (en utilisant le module `random`).

```

def hasard_couleur():
    R,V,B = randint(0,255),randint(0,255),randint(0,255)
    couleur = '#%02x'%02x'%02x' % (R%256, V%256, B%256)
    return couleur

```

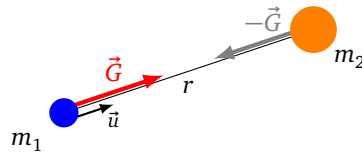
Une autre idée est de garder la trace des positions précédentes.



Cours 2 (Attraction gravitationnelle).

Deux astres s'attirent selon la force d'attraction gravitationnelle :

$$\vec{G} = G \frac{m_1 m_2}{r^2} \vec{u}$$



où

- les astres sont de masses m_1 et m_2 ,
- r est la distance entre les deux astres,
- G est la constante de gravitation universelle (on prendra arbitrairement $G = 100$),
- et $\vec{u}$ est un vecteur unité pointant de l'astre 1 vers l'astre 2.

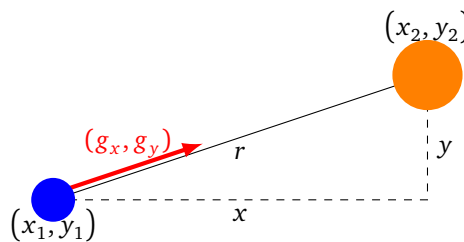
Avec ces notations $\vec{G}$ est la force d'attraction de l'astre 2 agissant sur l'astre 1, et $-\vec{G}$ est la force d'attraction de l'astre 1 agissant sur l'astre 2.

On note (x_1, y_1) , (x_2, y_2) les coordonnées des astres. On pose :

$$x = x_2 - x_1 \quad y = y_2 - y_1 \quad r = \sqrt{x^2 + y^2}$$

Les coordonnées du vecteur $\vec{G}$ sont alors :

$$g_x = G \frac{m_1 m_2}{r^2} \cdot \frac{x}{r} \quad \text{et} \quad g_y = G \frac{m_1 m_2}{r^2} \cdot \frac{y}{r}$$

**Activité 3** (Mouvement des planètes).

Objectifs : simuler le mouvement des planètes autour du Soleil grâce à la force de gravitation.

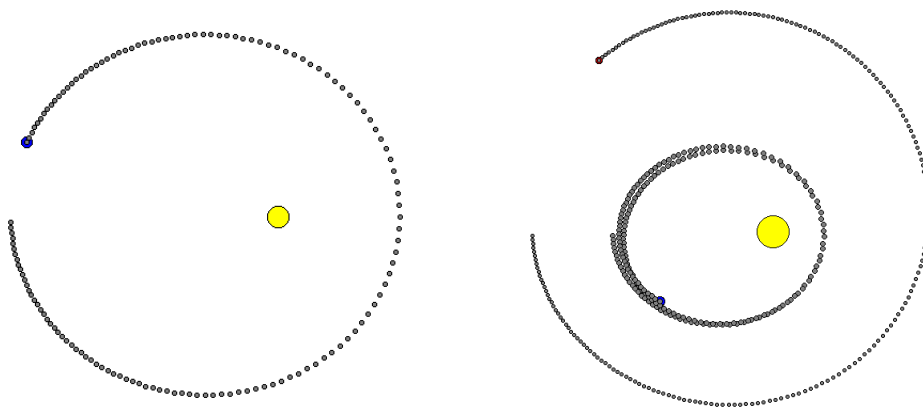
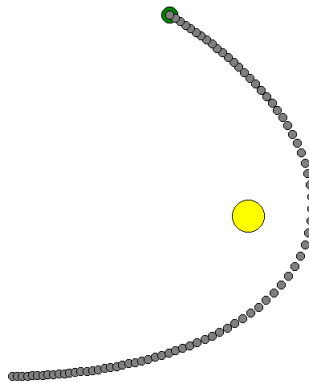


Figure de gauche : la Terre (en bleu) et le Soleil. Le Soleil est fixe. La Terre parcourt une orbite elliptique. Note que lorsqu'elle se rapproche du Soleil la Terre va plus vite.

Figure de droite : Mars (en rouge à l'extérieur), la Terre (en bleu) et le Soleil. Note que le mouvement de la Terre est perturbé par Mars. Dans la réalité le mouvement de la Terre est une ellipse quasiment circulaire et Mars influe peu sur le mouvement de la Terre (car sa masse est considérablement plus faible que celle du Soleil).

On peut aussi tracer la trajectoire d'une comète, ayant une vitesse initiale importante, qui passerait une unique fois près du Soleil avant de s'éloigner définitivement. Elle suit une trajectoire hyperbolique (figure ci-dessous).



Travail à faire :

1. Programme une classe `Planete` héritée de la classe `Particule`. Cette classe contient deux méthodes :
 - une méthode `action_attraction(self, other)` qui calcule la force d'attraction (g_x, g_y) entre l'astre courant (l'objet `self`) et un autre astre (l'objet `other`) et qui modifie la vitesse de l'astre courant (de masse m_1) selon la formule :

$$\begin{cases} v'_x &= v_x + \frac{g_x}{m_1} \\ v'_y &= v_y + \frac{g_y}{m_1} \end{cases}$$

- une méthode `mouvement(self)` qui ne fait appel qu'à la méthode `action_vitesse()`.
2. Programme une classe `TkPlanete` héritée de la classe `Planete` (de la même façon que `TkParticule()` était héritée de la classe `Particule()`) avec une méthode `affiche()` qui réalise l'affichage graphique avec `tkinter`.
 3. Utilise ton programme pour tracer les orbites de planètes. Par exemple avec trois astres (Soleil, Terre, Mars), à chaque pas il faut calculer l'attraction entre la Terre et le Soleil, l'attraction entre la Terre et Mars, puis déplacer la Terre. Ensuite il faut faire la même chose avec Mars. On considère que le Soleil est fixe.

Voici des exemples de paramètres utilisés (avec en plus $G = 100$).

```
# Trois astres : Soleil, Terre et Mars
soleil = TkPlanete(0,0,0,0,100,"yellow")
terre = TkPlanete(-200,0,0,-5,3,"blue")
mars = TkPlanete(-300,0,0,-5,2,"red")
```

```
for k in range(200):
    terre.action_attraction(soleil)
    terre.action_attraction(mars)
    terre.mouvement()
    terre.affiche(avec_trace=True)
    mars.action_attraction(soleil)
    mars.action_attraction(terre)
    mars.mouvement()
```



```
mars.affiche(avec_trace=True)  
  
canvas.update()  
sleep(0.02)
```


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 récurrents.

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 :

$$S_n = 1^2 + 2^2 + 3^2 + \dots + (n-1)^2 + n^2.$$

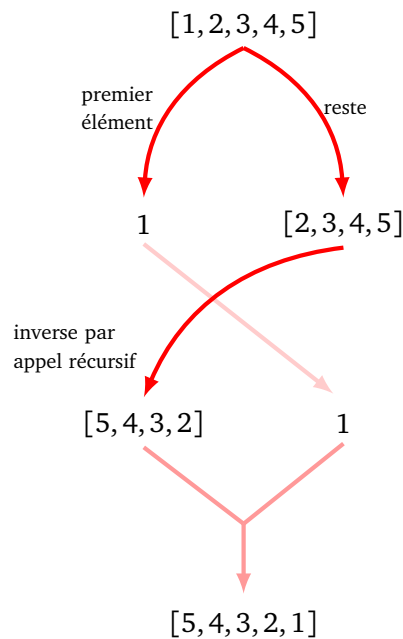
- (a) Programme une fonction `somme_carres_classique(n)` qui calcule cette somme S_n à l’aide d’une boucle.

- (b) En utilisant la formule de récurrence :

$$S_1 = 1 \quad \text{et} \quad S_n = S_{n-1} + n^2 \quad \text{pour } n \geq 2,$$

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 $n \geq 2$, 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.

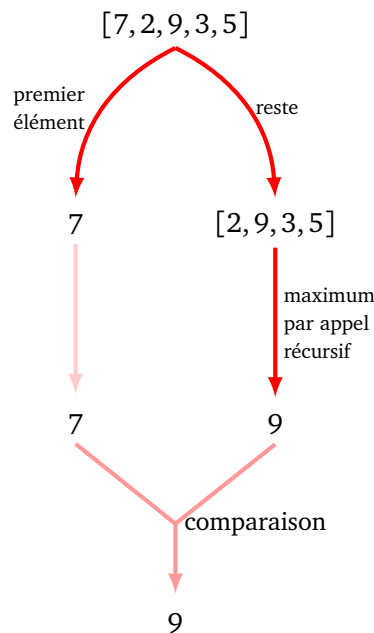


Voici l'algorithme en détails :

Algorithme.

- — Entête : `inverser(liste)`
 - Entrée : une liste $[x_0, x_1, \dots, x_{n-1}]$ de longueur n .
 - Sortie : la liste inversée $[x_{n-1}, x_{n-2}, \dots, x_1, x_0]$.
 - 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 le 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 $\rightarrow$ ERPE $\rightarrow$ RP donc « Faux »

ELLE $\rightarrow$ LL $\rightarrow$ " " 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.

$$F_0 = 0 \quad \text{et} \quad F_1 = 1 \quad \text{puis} \quad F_n = F_{n-1} + F_{n-2} \quad \text{pour } n \geq 2.$$

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.

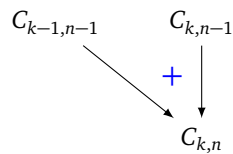
Commentaires. Cet algorithme n'est pas efficace car il est très lent pour $n \geq 30$. Essaie de comprendre pourquoi. Pour répondre à cette question tu peux afficher « Tiens, je calcule encore F_2 ! » à chaque appel de la fonction pour lequel $n = 2$.

2. Coefficients du binôme.

Les **coefficients du binôme de Newton** $C_{k,n}$ sont définis pour $0 \leq k \leq n$. 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

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

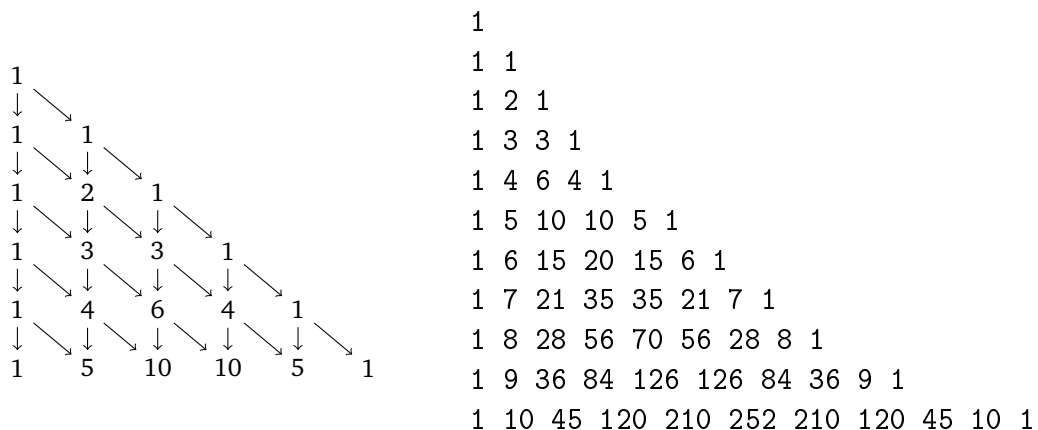
Programme une fonction récursive `binome(k, n)` qui renvoie $C_{k,n}$.

- 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 $k = 0, 1, \dots, n$ (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.



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

- 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 $r = 3$.

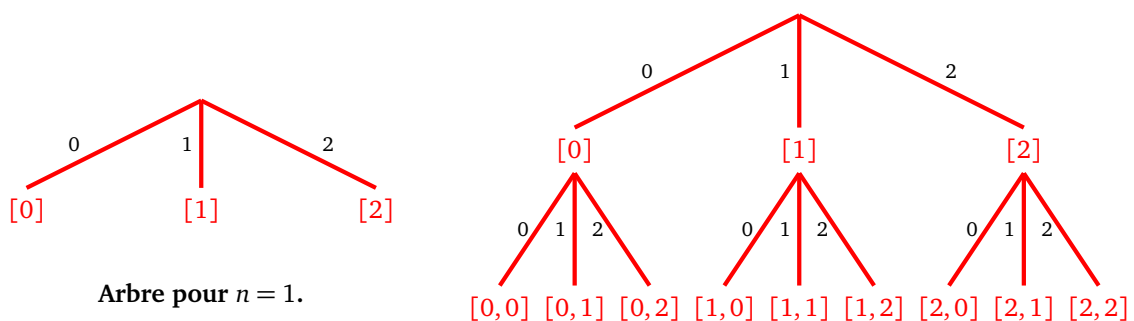
Programme une fonction récursive `residu_chiffres(n)` qui calcule le résidu r de l'entier n .

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.

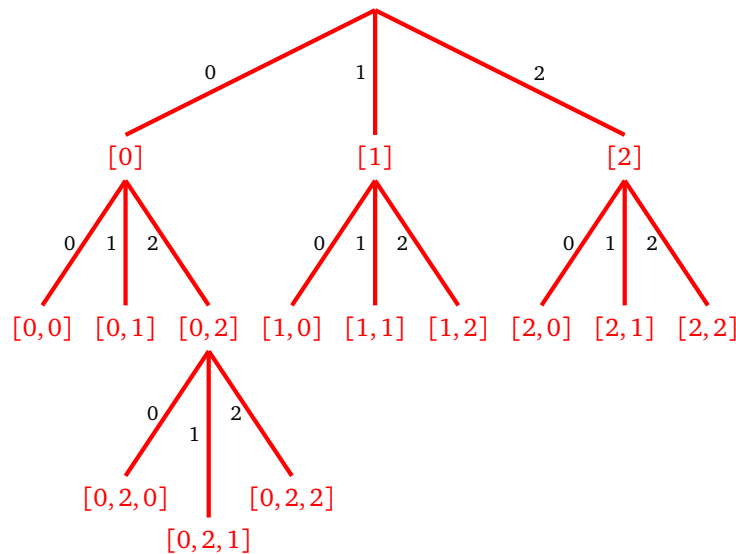
- Si $n = 1$, il n'y a que trois listes possibles d'un seul élément chacune : $[0]$, $[1]$, $[2]$.
- Si $n = 2$, il n'y a que 9 listes possibles ayant deux éléments : $[0, 0]$, $[0, 1]$, $[0, 2]$, $[1, 0]$, $[1, 1]$, $[1, 2]$, $[2, 0]$, $[2, 1]$, $[2, 2]$.
- On modélise cela sous la forme d'un arbre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

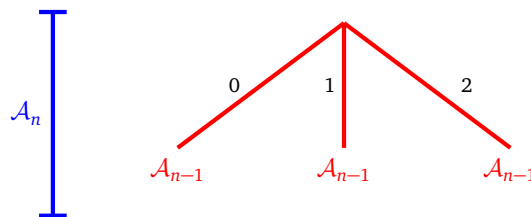


Une partie de l'arbre pour $n = 3$.

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 : $[[0,0,0],[0,0,1],[0,0,2],[0,1,0],[0,1,1], \dots, [2,2,1],[2,2,2]]$.

Activité 3 (Parcours d'arbre).

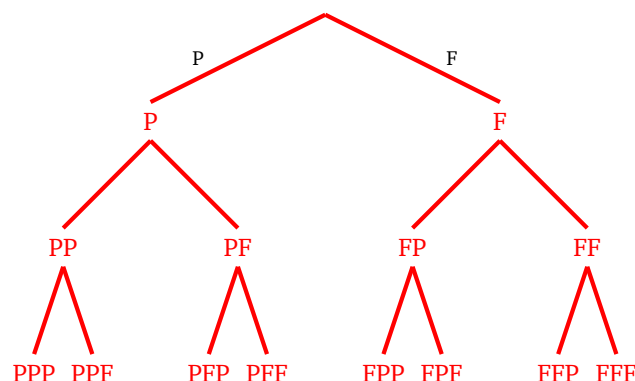
Objectifs : résoudre des problèmes en parcourant des arbres.

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...

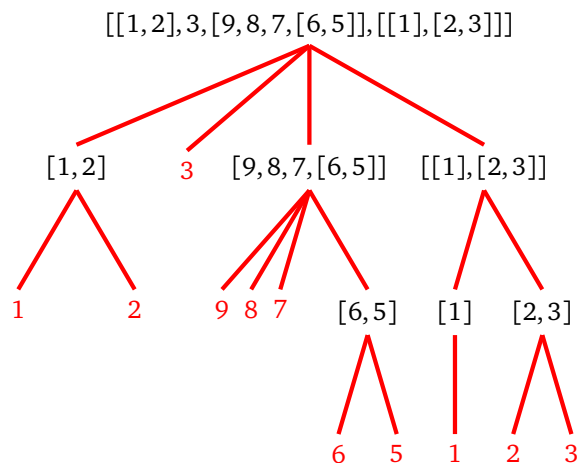
Par exemple :

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

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 :

$$p(a, b) = p(a, a + k/2) \cdot p(a + k/2, b) \quad \text{où } k = b - a.$$

Sur notre exemple cela revient à décomposer le produit en deux sous-produits :

$$p(10, 16) = (10 \times 11 \times 12) \times (13 \times 14 \times 15) = p(10, 13) \times p(13, 16).$$

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.

$W_0 \ W_1 \ W_2 \ W_3 \ W_4$
 $\downarrow \downarrow \downarrow \downarrow \downarrow$
 $S_3 \ S_4 \ S_2 \ S_1 \ S_0$

Permutation associée.

$0 \ 1 \ 2 \ 3 \ 4$
 $\downarrow \downarrow \downarrow \downarrow \downarrow$
 $3 \ 4 \ 2 \ 1 \ 0$

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 $i \mapsto j$.

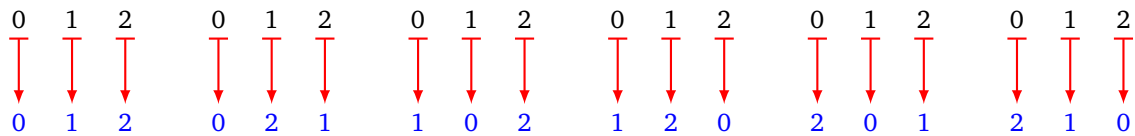
W_i
 $\downarrow$
 S_j

=

i
 $\downarrow$
 j

Permutation et dérangement.

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



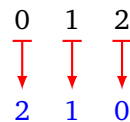
- Voici deux exemples avec $n = 5$:



- Un **dérangement** est une permutation qui vérifie $i \neq j$, pour l'association $i \mapsto j$ et ce quel que soit $i = 0, \dots, n-1$.
- 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 :

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

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 $e = \exp(1) = 2.718\dots$).
- Est-ce que la convergence est rapide (quand $n \rightarrow +\infty$) ?
- 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 :

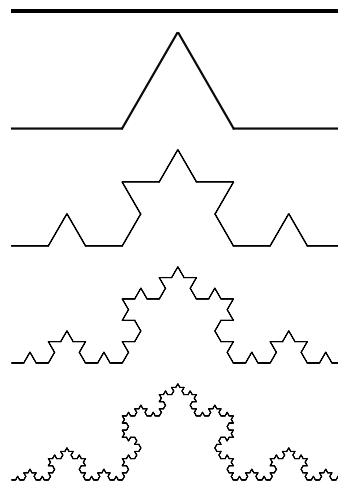
- `forward(100)/backward(100)` avancer/reculer de 100 pixels,
- `left(90)/right(90)` tourner à gauche/droite de 90 degrés,
- `goto(x,y)` aller à la position (x,y) ,
- `x,y = position()` récupérer les coordonnées courantes de la tortue,
- `setheading(angle)` s'orienter dans la direction donnée,
- `up()/down()` 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 $\text{koch}(1, n)$ qui trace le flocon de Koch d'ordre n ; ℓ est un paramètre de longueur.

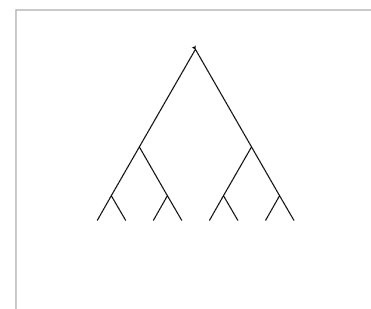
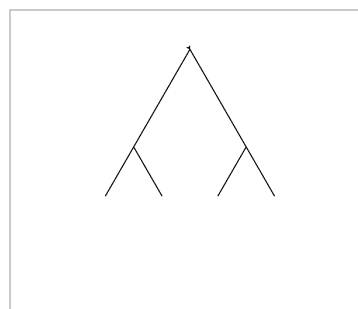
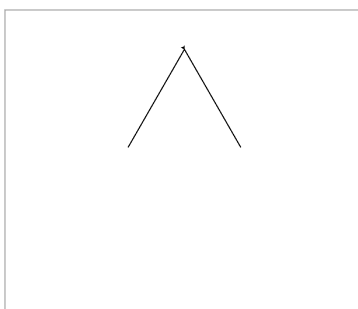
Le principe du tracé est le suivant :

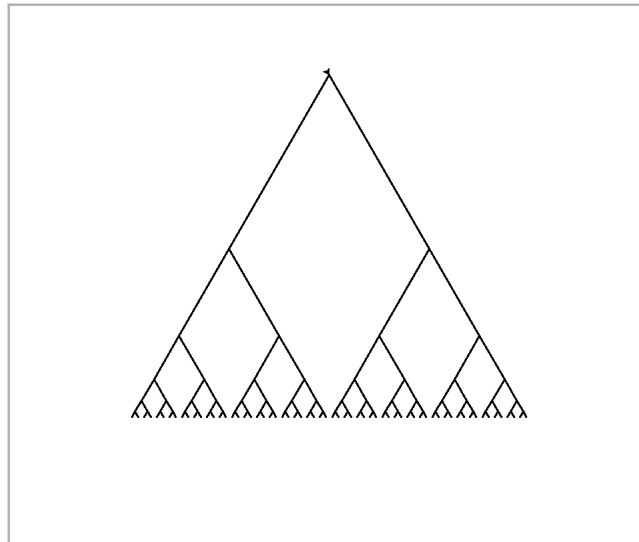
- *Cas terminal.* Si $n = 0$, tracer un segment de longueur ℓ .
- *Cas général.*
 - Tracer le flocon de Koch d'ordre $n - 1$, associé à la longueur $\ell/3$.
 - Tourner un peu vers la gauche.
 - Tracer le flocon de Koch d'ordre $n - 1$, associé à la longueur $\ell/3$.
 - Tourner vers la droite.
 - Tracer 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 $\text{arbre}(1, n)$ 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 $n = 6$.

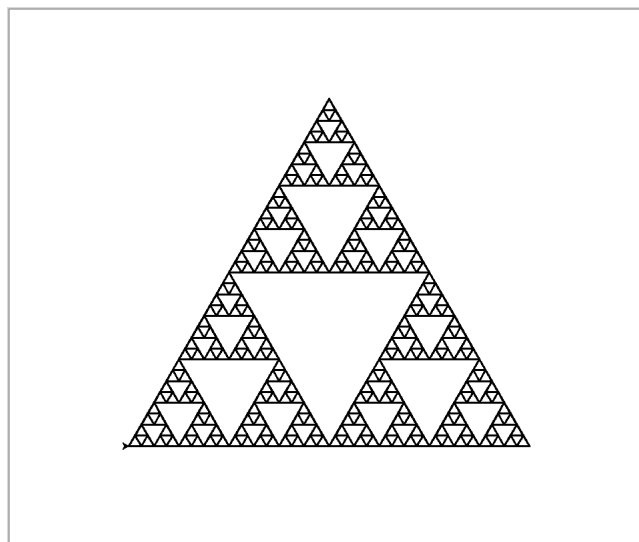
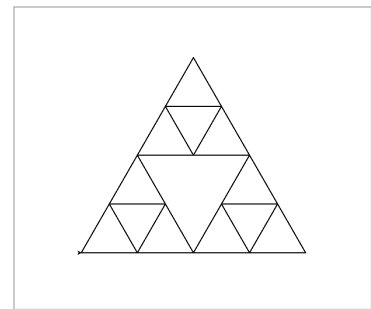
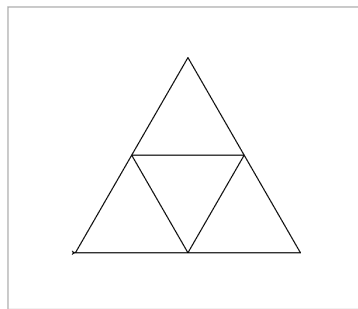
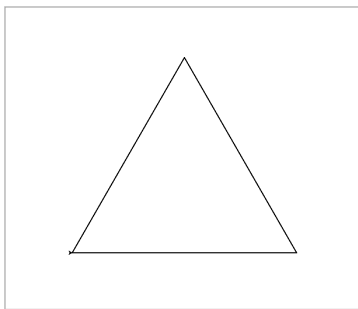




3. Triangle de Sierpinski.

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 :

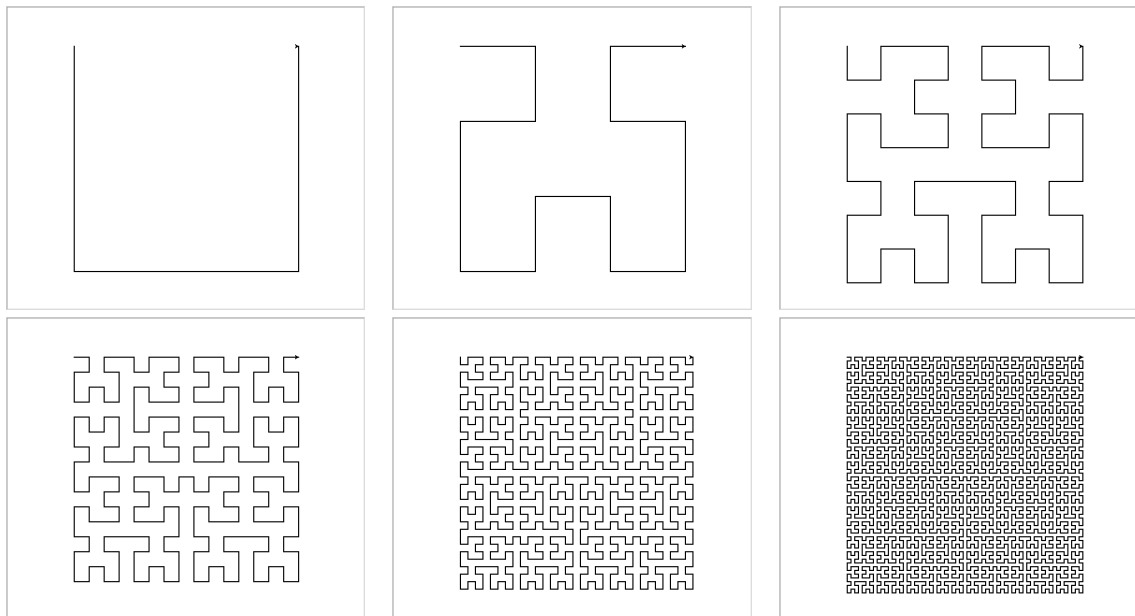
- Si $n = 0$ ne rien faire.
- Sinon répéter trois fois :
 - faire un appel récursif `triangle(l/2,n-1)`,
 - avancer de ℓ pas,
 - tourner de 120 degrés.

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 $-\theta$.

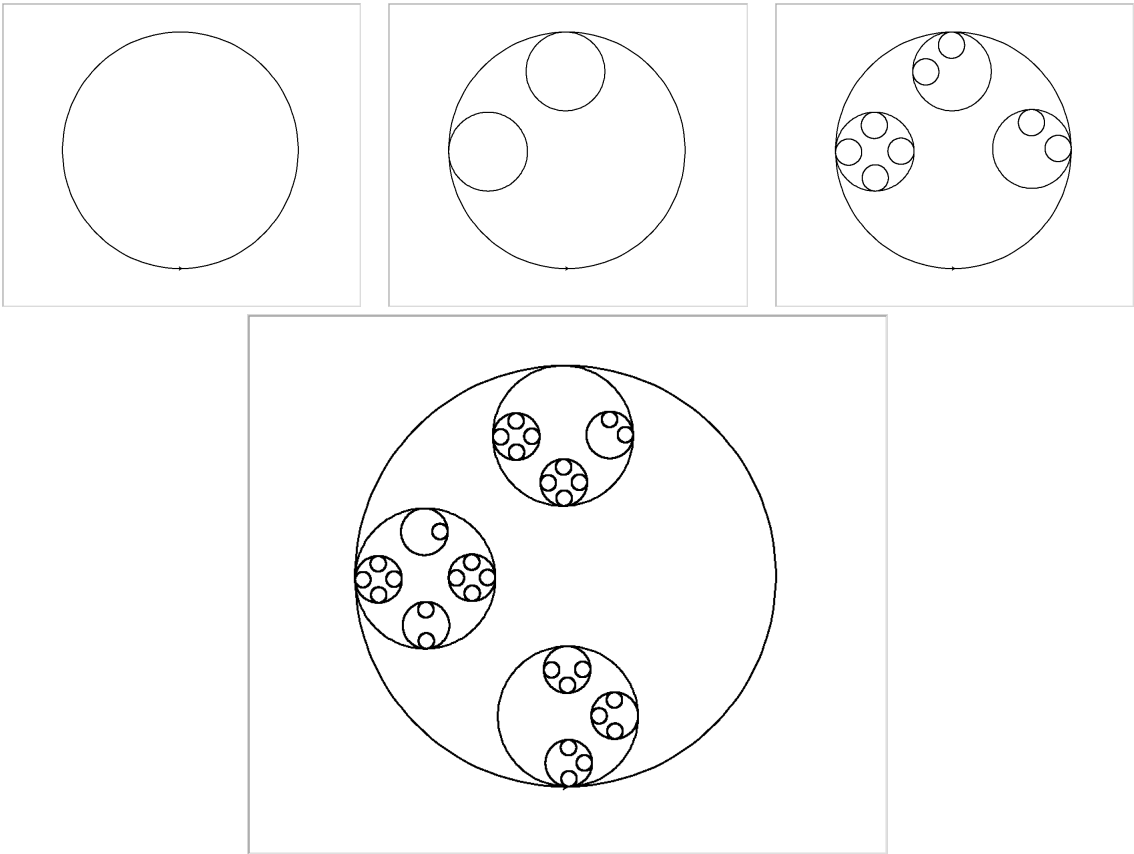
Voici les dessins pour l'angle initial valant $\theta = +90$ degré et des ordres n allant de 1 à 6.



5. Fractale aléatoire.

Programme une fonction récursive `fractale_cercle(1, 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.



Tri – Complexité

Ordonner les éléments d'une liste est une activité essentielle en informatique. Par exemple une fois qu'une liste est triée, il est très facile de chercher si elle contient tel ou tel élément. Par définition un algorithme renvoie toujours le résultat attendu, mais certains algorithmes sont plus rapides que d'autres ! Cette efficacité est mesurée par la notion de complexité.

Ce chapitre commence par de la théorie : tout d'abord des rappels sur les suites et l'explication de la notation « grand O ». Ensuite on aborde la notion de complexité qui mesure la performance d'un algorithme. Ceux qui veulent coder peuvent directement s'attaquer aux différents algorithmes de tris présentés. Le bilan est fait dans la dernière activité : comparer les complexités des différents algorithmes de tris.

Cours 1 (Notation « grand O »).

On souhaite comparer deux suites, ou plus exactement leur ordre de grandeur. Par exemple les suites $(n^2)_{n \in \mathbb{N}}$ et $(3n^2)_{n \in \mathbb{N}}$ ont le même ordre de grandeur, mais sont beaucoup plus petites que la suite $(\frac{1}{2}e^n)_{n \in \mathbb{N}}$.

Notation « grand O ».

- On considère $(u_n)_{n \in \mathbb{N}}$ et $(v_n)_{n \in \mathbb{N}}$ deux suites de termes strictement positifs.
- On dit que (u_n) est un **grand O** de (v_n) si la suite $(\frac{u_n}{v_n})$ est bornée.
- Autrement dit il existe une constante réelle $k > 0$ telle que pour tout $n \in \mathbb{N}$:

$$u_n \leq k v_n.$$

- *Notation.* On note alors $u_n = O(v_n)$. Il s'agit de la lettre « O » (pour Ordre de grandeur) et pas du chiffre zéro.

Exemples

- Soient $u_n = 3n + 1$ et $v_n = 2n - 1$. Comme $\frac{u_n}{v_n} \rightarrow \frac{3}{2}$ lorsque $n \rightarrow +\infty$ alors la suite $(\frac{u_n}{v_n})$ est bornée donc $u_n = O(v_n)$.
- $u_n = 2n^2$ et $v_n = e^n$. Comme $\frac{u_n}{v_n} \rightarrow 0$ alors la suite $(\frac{u_n}{v_n})$ est bornée donc $u_n = O(v_n)$.
- $u_n = \sqrt{n}$ et $v_n = \ln(n)$. Comme $\frac{u_n}{v_n} \rightarrow +\infty$ lorsque $n \rightarrow +\infty$ alors la suite $(\frac{u_n}{v_n})$ n'est pas bornée. (u_n) n'est pas un grand O de (v_n) . Par contre dans l'autre sens, on a bien $v_n = O(u_n)$.
- $u_n \in O(n)$ signifie qu'il existe $k > 0$ tel que $u_n \leq kn$ (pour tout $n \in \mathbb{N}$).
- $u_n \in O(1)$ signifie que la suite (u_n) est bornée.

Suites de référence.

On va de préférence comparer une suite (u_n) avec des suites de référence. Voici les suites de référence choisies :

$$\underbrace{\ln(n)}_{\text{croissance logarithmique}} \quad \underbrace{n \quad n^2 \quad n^3 \quad \dots}_{\text{croissance polynomiale}} \quad \underbrace{e^n}_{\text{croissance exponentielle}}$$

- Les suites sont écrites en respectant l'ordre des O : on a $\ln(n) = O(n)$, $n = O(n^2)$, $n^2 = O(n^3)$, ..., $n^3 = O(e^n)$.

- On pourrait intercaler d'autres suites, par exemple $\ln(n) = O(\sqrt{n})$ et $\sqrt{n} = O(n)$. Ou encore $n \ln(n) = O(n^2)$.
- Il est important de savoir visualiser ces suites (voir le graphique de l'activité 1).

Activité 1 (Notation « grand O »).

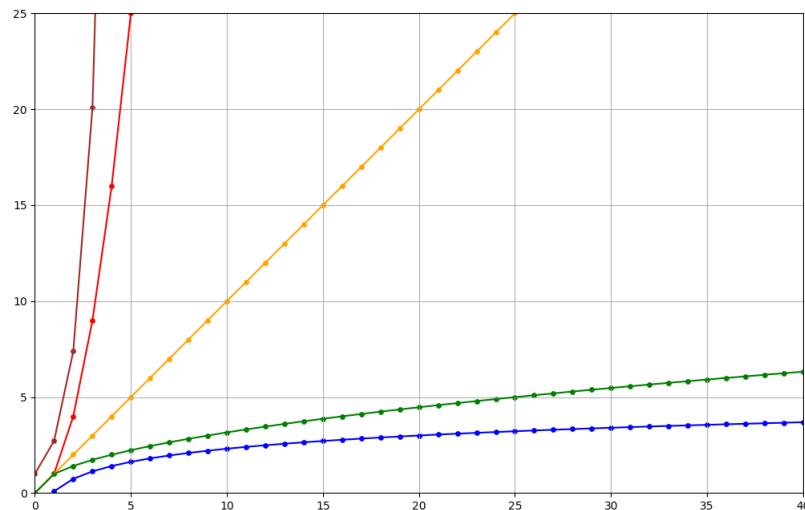
Objectifs : comparer des suites avec la notation « grand O ».

1. Considère les suites définies par

$$u_n = 1000n^2 \quad \text{et} \quad v_n = 0.001 \exp(n)$$

Calcule les premiers termes de chaque suite. Penses-tu que $u_n = O(v_n)$ ou bien $v_n = O(u_n)$?

2. Visualise les termes de différentes suites, comme sur le graphique ci-dessous où tu retrouves les termes des suites $\ln(n)$, $\sqrt{n}$, n , n^2 , e^n .



Les suites $\ln(n)$, $\sqrt{n}$, n , n^2 , e^n .

3. Programme une fonction `est_grand_O(u, v)` qui renvoie « Vrai » si la suite (u_n) est expérimentalement un grand O de (v_n) . On dira que (u_n) est expérimentalement un grand O de (v_n) si

$$u_n \leq k v_n$$

pour $n \in [10, 1000]$ et $k = 10$. (Bien sûr le choix de ces constantes est arbitraire.)

A-t-on expérimentalement $n^2 = O(2^n)$? Et $n = O(\sqrt{n})$?

Tu peux définir une suite u comme une fonction :

```
def u(n): return n**2
```

ou bien

```
u = lambda n: n**2
```

Dans les deux cas on obtient le terme u_n par la commande `u(n)`.

Cours 2 (Complexité d'un algorithme).

On mesure l'efficacité d'un algorithme à l'aide de la complexité.

- Les deux principales caractéristiques qui font qu'un algorithme est bon ou mauvais sont la rapidité d'exécution et l'utilisation de la mémoire. Nous nous limiterons ici à étudier la vitesse d'exécution.

- Comment mesurer la vitesse ? Une durée (en secondes) dépend de chaque ordinateur et n'est pas un indicateur universel.
- Aussi nous définissons de manière informelle la complexité : la **complexité** d'un algorithme est le nombre d'opérations élémentaires exécutées.
- Ce que l'on appelle « opération élémentaire » peut varier selon le contexte : pour un calcul cela peut être le nombre de multiplications, pour un tri le nombre de comparaisons...
- La complexité C_n dépend de la taille n des données en entrée (par exemple le nombre de chiffres d'un entier ou bien la longueur de la liste). On obtient ainsi une suite (C_n) .
- Les bons algorithmes ont des complexités polynomiales qui sont en $O(n)$ (linéaire), ou en $O(n^2)$ (quadratique) ou bien en $O(n^k)$, $k \in \mathbb{N}^*$ (polynomiale). Les mauvais algorithmes ont des complexités exponentielles, en $O(e^n)$ par exemple.

Multiplication de deux entiers.

On souhaite multiplier deux entiers a et b de n chiffres. Il y a plusieurs méthodes, on les compare en comptant le nombre d'opérations élémentaires : ici des multiplications de petits nombres (entiers à 1 ou 2 chiffres).

Algorithme	Ordre de la complexité
Multiplication d'école	$O(n^2)$
Multiplication de Karatsuba	$O(n^{\log_2(3)}) \simeq O(n^{1.53})$
Transformée de Fourier rapide	$O(n \cdot \ln(n) \cdot \ln(\ln(n)))$

Voici des exemples d'ordre de grandeur de la complexité pour différentes valeurs de n .

Algorithme	$n = 10$	$n = 100$	$n = 1000$
Multiplication d'école	100	10 000	1 000 000
Multiplication de Karatsuba	38	1478	56 870
Transformée de Fourier rapide	19	703	13 350

Plus l'entier n est grand, plus un bon algorithme prend l'avantage.

Recherche dans une liste.

La recherche d'un élément dans une liste non triée nécessite de tester chaque élément de la liste. Si la liste est de longueur n alors il faut $O(n)$ tests. Par contre si la liste est ordonnée alors il existe des algorithmes beaucoup plus efficaces : par exemple la recherche par dichotomie (voir le chapitre « Le mot le plus long »).

Algorithme	Ordre de la complexité
Élément par élément (liste non triée)	$O(n)$
Dichotomie (liste triée)	$O(\log_2(n))$

Voici des exemples d'ordre de grandeur de la complexité pour différentes valeurs de n .

Algorithme	$n = 1000 = 10^3$	$n = 10^6$	$n = 10^9$
Élément par élément	10^3	10^6	10^9
Dichotomie	10	20	30

Problème du voyageur de commerce.

On se donne n villes et les distances entre ces villes. Il s'agit de trouver le plus court chemin qui visite toutes les villes en revenant à la ville de départ. Il n'y a pas d'algorithme connu qui soit efficace pour obtenir

la meilleure solution. Un des meilleurs algorithmes a pour complexité $O(n^2 2^n)$. Voici des exemples d'ordre de grandeur de la complexité pour différentes valeurs de n .

Algorithme	$n = 10$	$n = 100$	$n = 1000$
Voyageur de commerce	10^5	10^{34}	10^{307}

On voit que cet algorithme est inutilisable sauf pour de petites valeurs de n .

Cours 3 (Le tri avec Python).

- La commande Python pour ordonner une liste est `sorted()`. Par exemple avec `liste = [5, 6, 1, 8, 10]`, la commande `sorted(liste)` renvoie la nouvelle liste `[1, 5, 6, 8, 10]` dans laquelle les éléments sont ordonnés du plus petit au plus grand.
Cela fonctionne aussi avec des chaînes de caractères, pour

```
liste = ['BATEAU', 'ABRIS', 'ARBRE', 'BARBE']
```

alors `sorted(liste)` renvoie `['ABRIS', 'ARBRE', 'BARBE', 'BATEAU']` ordonnée selon l'ordre alphabétique.
- Variante. La méthode `liste.sort()` ne renvoie rien, mais après utilisation de cette méthode, `liste` est ordonnée (on parle de modification en place).
- Pour obtenir un tri dans l'ordre inverse, utilise la commande `sorted(liste, reverse = True)`.
- Variante. `list(reversed(sorted(liste)))`.

Cours 4 (Double affectation avec Python).

Python permet les affectations multiples, ce qui permet d'échanger facilement le contenu de deux variables.

- **Affectation multiple.**

```
a, b = 3, 4
```

Maintenant `a` vaut 3 et `b` vaut 4.

- **Échange de valeurs.**

```
a, b = b, a
```

Maintenant `a` vaut l'ancien contenu de `b` donc vaut 4 et `b` vaut l'ancien contenu de `a` donc 3.

- **Échange à la main.** Pour échanger deux valeurs sans utiliser la double affectation, il faut introduire une variable temporaire :

```
temp = a
a = b
b = temp
```

Activité 2 (Tri par sélection).

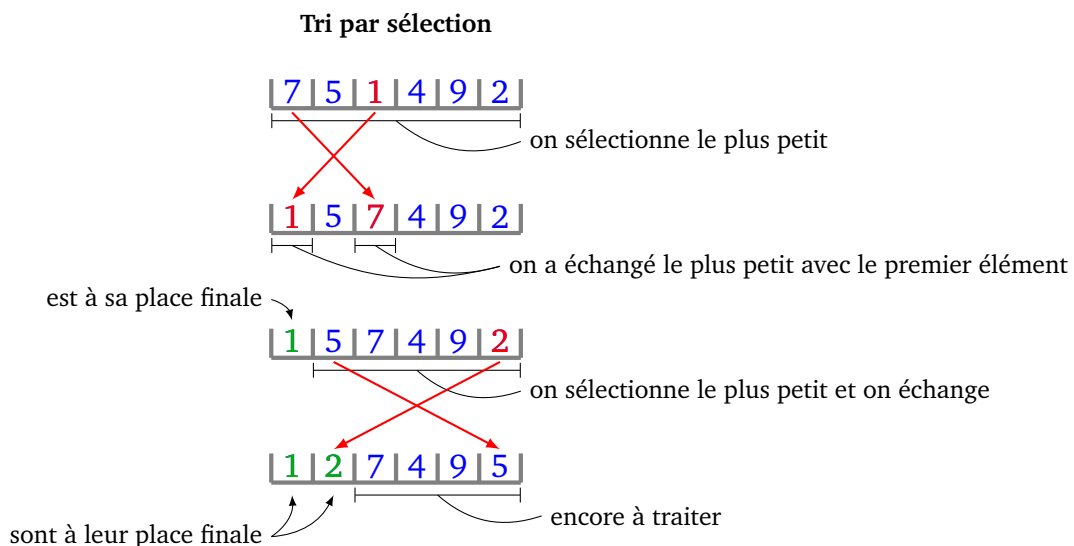
Objectifs : programmer le « tri par sélection » qui est un algorithme très simple.

Il s'agit d'ordonner les éléments d'une liste du plus petit au plus grand. On note n la longueur de la liste. Les éléments sont donc indexés de 0 à $n - 1$.

Algorithme.

- — Entrée : une liste de longueur n .
— Sortie : la liste ordonnée.
- Pour i variant de 0 à $n - 1$:
— Recherche du plus petit élément après le rang i :
— $rg_min \leftarrow i$
— pour j allant de $i + 1$ à $n - 1$:
 si $liste[j] < liste[rg_min]$ faire $rg_min \leftarrow j$.
— Échange. Échanger l'élément de rang i avec l'élément de rang rg_min .
- Renvoyer la liste.

Explications. L'algorithme est très simple : on cherche le plus petit élément de la liste et on le place en tête. Le premier élément est donc à sa place. On recommence avec le reste de la liste : on cherche le plus petit élément que l'on positionne en deuxième place...



Travail à faire. Programme cet algorithme en une fonction `tri_selection(liste)`.

Indications. La fonction ne doit pas modifier la liste passée en paramètre. Pour éviter les désagréments :

- commence par faire une copie de ta liste :
`cliste = list(liste)`
- ne travaille qu'avec `cliste`, que tu peux modifier à volonté,
- renvoie `cliste`.

Commentaires. Le principal avantage de cet algorithme est sa simplicité. Sinon il est de complexité $O(n^2)$ ce qui en fait un algorithme de tri lent réservé pour les petites listes.

Activité 3 (Tri par insertion).

Objectifs : programmer le « tri par insertion » qui est un algorithme très simple.

Le tri par insertion est assez naturel : c'est le tri que tu utilises par exemple pour trier un jeu de cartes. Tu prends les deux premières cartes, tu les ordonnes. Tu prends la troisième carte, tu la places au bon

endroit pour obtenir trois cartes bien ordonnées. Tu prends une quatrième carte que tu places au bon endroit pour obtenir quatre cartes bien ordonnées...

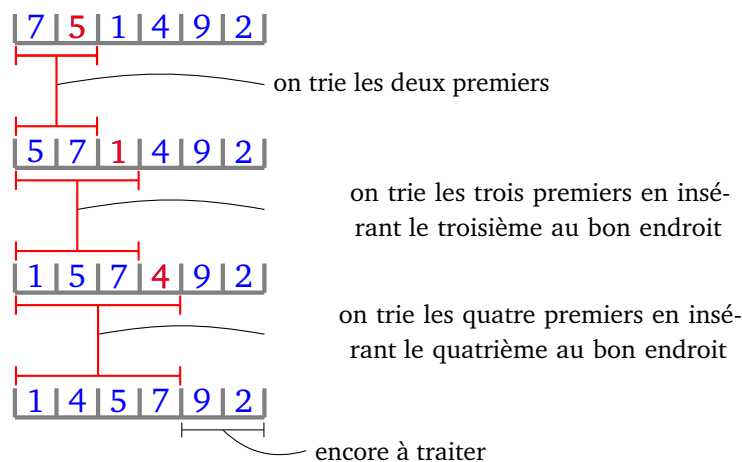
Algorithme.

- — Entrée : une liste de longueur n .
- Sortie : la liste ordonnée.
- Pour i variant de 1 à $n - 1$:
 - $el \leftarrow \text{liste}[i]$ (on mémorise l'élément pivot de rang i)
 - On décale vers la droite tous les éléments de rang $i - 1$ à 0 qui sont plus grands que le pivot :
 - $j \leftarrow i$
 - Tant que $(j > 0)$ et $(\text{liste}[j-1] > el)$:
 - $\text{liste}[j] \leftarrow \text{liste}[j-1]$
 - $j \leftarrow j - 1$
 - $\text{liste}[j] \leftarrow el$ (on remplace l'élément pivot dans le trou créé par le décalage)
- Renvoyer la liste.

Explications. L'algorithme est assez simple : on regarde les deux premiers éléments, s'ils sont dans le mauvais sens on les échange (les deux premiers éléments sont maintenant bien ordonnés entre eux). On regarde ensuite les trois premiers éléments, on insère le troisième élément à la bonne place parmi ces trois éléments (qui sont maintenant bien ordonnés entre eux). On recommence avec les quatre premiers éléments : on insère le quatrième élément à la bonne place parmi ces quatre éléments...

Le dernier élément du groupe considéré est appelé pivot, pour l'insérer on décale d'un rang vers la droite tous les éléments situés avant lui qui sont plus grands. On obtient donc un trou qui est la place du pivot.

Tri par insertion



Travail à faire. Programme cet algorithme en une fonction `tri_insertion(liste)`.

Commentaires. C'est un bon algorithme dans la catégorie des algorithmes de tri lents ! Il est de complexité $O(n^2)$, mais sur une liste déjà un peu ordonnée il est efficace. Il est un peu meilleur que le tri par sélection. Il permet aussi d'ordonner des éléments en « temps réels » : on peut commencer à trier le début de la liste sans connaître la fin.

Activité 4 (Tri à bulles).

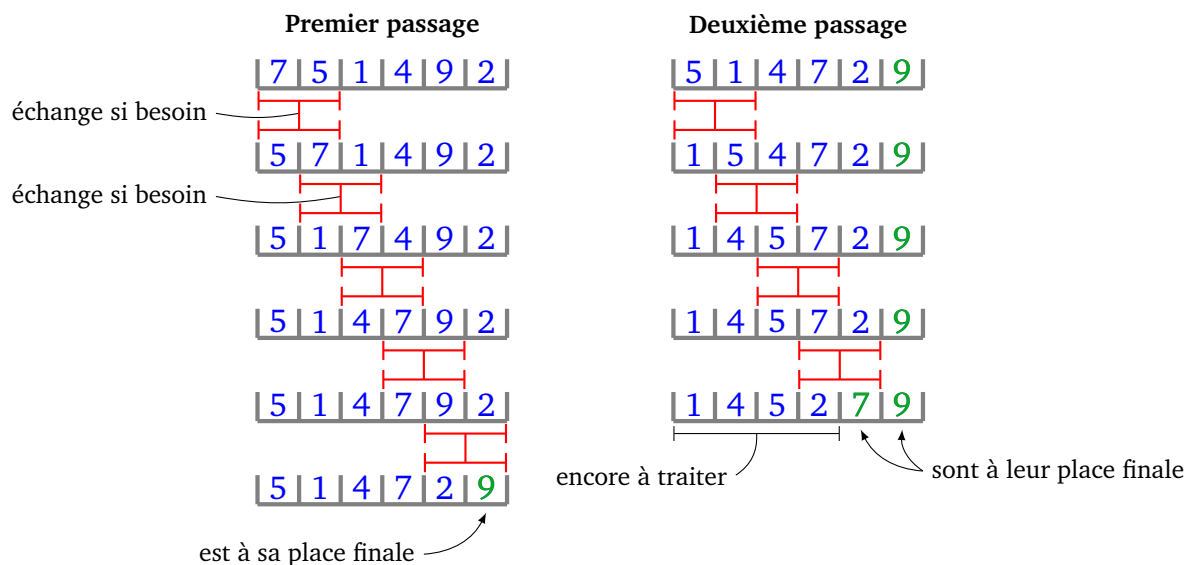
Objectifs : programmer le « tri à bulles ».

Le tri à bulles est très simple à programmer : il s'agit d'échanger deux termes consécutifs s'ils ne sont pas dans le bon ordre. Le nom vient de l'analogie avec les bulles d'eau qui remontent à la surface comme ici les éléments qui viennent se positionner à leur place.

Algorithme.

- — Entrée : une liste de longueur n .
- Sortie : la liste ordonnée.
- Pour i allant de $n - 1$ à 0 :
 - Pour j allant de 0 à $i - 1$:
 - Si `liste[j+1] < liste[j]` alors :
 - échanger `liste[j]` et `liste[j+1]`.
- Renvoyer la liste.

Explications. L'algorithme est très simple : on compare deux éléments consécutifs et on les échange s'ils sont dans le mauvais ordre. On continue avec les couples suivants jusqu'à la fin de la liste. Au bout du premier passage le dernier élément est définitivement à sa place. On recommence en partant du début avec un second passage, maintenant les deux derniers éléments sont à leur place.

Tri à bulles

Travail à faire. Programme cet algorithme en une fonction `tri_a_bulles(liste)`.

Commentaires. Le tri à bulles est très simple à programmer, cependant il fait aussi partie des algorithmes de tri lents car sa complexité est en $O(n^2)$.

Activité 5 (Tri fusion).

Objectifs : programmer un tri beaucoup plus efficace : le « tri fusion ». Par contre sa programmation est plus compliquée car l'algorithme est récursif.

Le tri fusion est un tri rapide. Il est basé sur le principe de « diviser pour régner » ce qui fait que sa programmation naturelle se fait par une fonction récursive. Le principe est simple : on divise la liste en deux parties ; on trie la liste de gauche (par un appel récursif) ; on trie la liste de droite (par un autre appel récursif) ; ensuite il faut fusionner ces deux listes en intercalant les termes.

Ce tri se programme à l'aide de deux fonctions : une fonction principale `tri_fusion(liste)` qui trie la liste. Cette fonction nécessite la fonction secondaire `fusion(liste_gauche, liste_droite)` qui fusionne deux listes triées en une seule.

On commence par la fonction principale qui est une fonction récursive.

Algorithme.

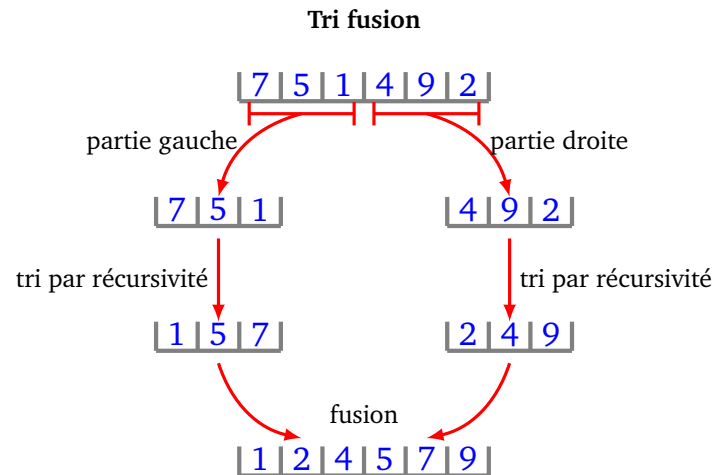
- — Entête : `tri_fusion(liste)`
- Entrée : une liste de longueur n .
- Sortie : la liste ordonnée.
- Action : fusion récursive.
- *Cas terminal.* Si la liste est de longueur 0 ou 1, renvoyer la liste telle quelle.
- *Cas général.*
- Calculer `liste_g = tri_fusion(liste[:n//2])`. On prend les éléments de gauche (de rang $< n//2$) et on les trie par un appel récursif.
- Calculer `liste_d = tri_fusion(liste[n//2:])`. On prend les éléments de droite (de rang $\geq n//2$) et on les trie par un appel récursif.
- Renvoyer la liste `fusion(liste_g, liste_d)`.

La fonction précédente nécessite la fonction définie par l'algorithme suivant :

Algorithme.

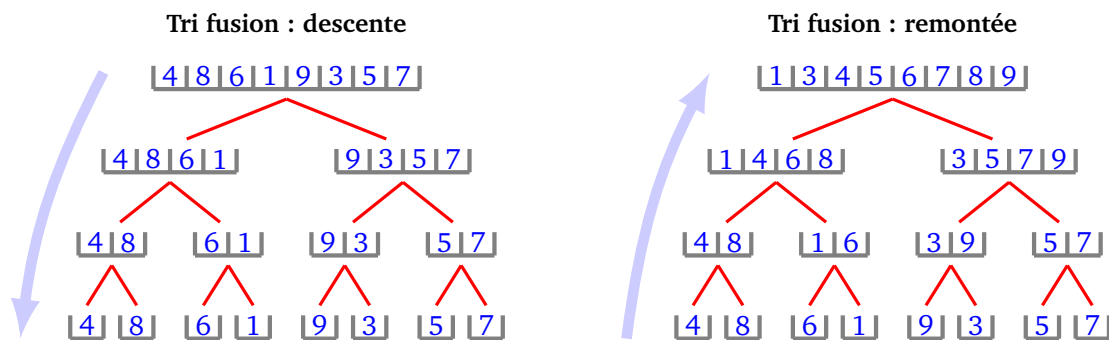
- — Entête : `fusion(liste_g, liste_d)`
- Entrée : deux listes ordonnées : `liste_g` de longueur n et `liste_d` de longueur m .
- Sortie : une liste fusionnée et ordonnée.
- — Un indice i , initialisé à 0, indexe la première liste,
- un indice j , initialisé à 0, indexe la seconde liste,
- une liste `liste_fus` est initialisée à la liste vide.
- *Fusion principale.*
- Tant que $(i < n)$ et $(j < m)$, faire :
 - si `liste_g[i] < liste_d[j]` ajouter `liste_g[i]` à `liste_fus` et incrémenter i ,
 - sinon ajouter `liste_d[j]` à `liste_fus` et incrémenter j .
- *S'il reste des termes.*
- Tant que $i < n$, ajouter `liste_g[i]` à `liste_fus` et incrémenter i .
- Tant que $j < m$, ajouter `liste_d[j]` à `liste_fus` et incrémenter j .
- Renvoyer `liste_fus`.

Explications. La seconde fonction `fusion()` regroupe deux listes ordonnées en une seule liste ordonnée. La fonction `tri_fusion(liste)` est très simple, comme nous l'avons expliqué précédemment : la liste est découpée en une partie droite et une partie gauche. Grâce à deux appels récursifs chacune de ces sous-parties est triée. Il ne reste plus qu'à fusionner ces deux listes.



Bien sûr, comme toute fonction récursive, c'est assez difficile d'appréhender l'enchaînement complet des instructions.

Pourquoi cet algorithme est-il plus performant que les précédents ? Ce n'est pas facile à expliquer ! Le point-clé se passe lors la fusion : quand on fusionne deux listes ordonnées de longueur n on effectue des comparaisons $\text{liste_g}[i] < \text{liste_d}[j]$, mais on ne compare pas tous les éléments entre eux (ce qui donnerait n^2 comparaisons) mais seulement pour quelques couples (i, j) ce qui donne $2n$ comparaisons.



Travail à faire. Programme ces algorithmes en deux fonctions `fusion(liste_g, liste_d)` et `tri_fusion(liste)`.

Commentaires. Le tri fusion est un tri rapide : sa complexité est $O(n \ln(n))$ ce qui est beaucoup mieux que les algorithmes précédents et est asymptotiquement optimal. Il est conceptuellement simple à comprendre et à programmer si on connaît la récursivité.

Activité 6 (Complexité des algorithmes de tri).

Objectifs : comparer expérimentalement les complexités des algorithmes de tri.

Pour les quatre algorithmes de tri que tu as programmés, modifie tes fonctions afin qu'elles renvoient en plus de la liste triée le nombre de comparaisons effectuées entre deux éléments de la liste. Chaque test du type « `liste[i] < liste[j]` » compte pour une comparaison.

- Pour les trois premiers algorithmes, c'est assez facile ; pour le dernier c'est plus compliqué (voir ci-dessous).
- Compare le nombre de comparaisons effectuées :

- quand la liste est une liste d'éléments tirés au hasard ;
- quand la liste est déjà triée ;
- quand la liste est déjà triée mais en sens inverse.
- On note n la longueur de la liste. Compare la complexité avec n^2 (ou mieux $n^2/2$) pour les algorithmes lents et $n \ln(n)$ (ou mieux $n \log_2(n)$) pour l'algorithme de tri fusion.

Indications pour le tri fusion. Il faut commencer par modifier la fonction `fusion()` en une fonction `fusion_complexite()` qui renvoie en plus de la liste, le nombre de comparaisons effectuées lors de cette étape. Il faut ensuite modifier la fonction `tri_fusion()` en une fonction `tri_fusion_complexite()` qui renvoie en plus le nombre de comparaisons. Pour cela il faut compter le nombre de comparaisons venant du tri fusion de la partie gauche de la liste, le nombre de comparaisons venant du tri fusion de la partie droite de la liste, et enfin le nombre de comparaisons venant de la fusion. Il faut finalement renvoyer la somme de ces trois nombres.

Une jolie activité consiste à visualiser pas à pas le tri d'une liste pour chaque algorithme. Voir les pages Wikipédia pour un tel exemple d'animation.

Calculs en parallèle

Comment profiter d'avoir plusieurs processeurs (ou plusieurs cœurs dans chaque processeur) pour calculer plus vite ? C'est simple, il suffit de partager les tâches à réaliser afin que tout le monde travaille en même temps, puis de regrouper les résultats. Dans la pratique, ce n'est pas si facile.

Cours 1 (Calculs en parallèle : motivation).

Tu as trouvé un vaccin contre les zombies qui ravagent le monde. Comment distribuer le plus rapidement possible les pilules à 100 personnes, sachant qu'on ne peut être en contact qu'avec une seule personne à la fois ? Méthode séquentielle (un seul processeur) : distribuer les pilules une par une. S'il faut 1 minute pour chaque distribution, il te faudra en tout 100 minutes. Méthode à deux processeurs : donne la moitié des pilules à un camarade (1 minute), puis chacun distribue ses pilules. Au total cela prend environ 50 minutes donc deux fois plus rapide. Peux-tu faire mieux ?

Habituellement on effectue un calcul de façon séquentielle : le calcul est divisé en plusieurs tâches, chacune est exécutée tour à tour pour à la fin donner le résultat.

Dans le calcul en parallèle : on répartit les tâches entre plusieurs processeurs, qui calculent simultanément. Par exemple : comment calculer $7 + 5 + 9 + 4$? De façon séquentielle on fait $7 + 5 = 12$ (1 seconde) puis on fait $12 + 9 = 21$ (1 seconde), et enfin $21 + 4 = 25$, pour un total de trois secondes. On peut faire mieux avec deux processeurs : le premier calcule $7 + 5 = 12$, le second calcule $9 + 4 = 13$ (cela fait en tout une seule seconde), puis le premier calcule $12 + 13 = 25$. Au total cela a pris deux secondes.

$$\begin{array}{r}
 7 + 5 + 9 + 4 \\
 \hline
 12 + 9 + 4 \\
 \hline
 21 + 4 \\
 \hline
 25
 \end{array}$$

Calculs séquentiels

$$\begin{array}{r}
 7 + 5 + 9 + 4 \\
 \hline
 12 + 13 \\
 \hline
 25
 \end{array}$$

Calculs en parallèle

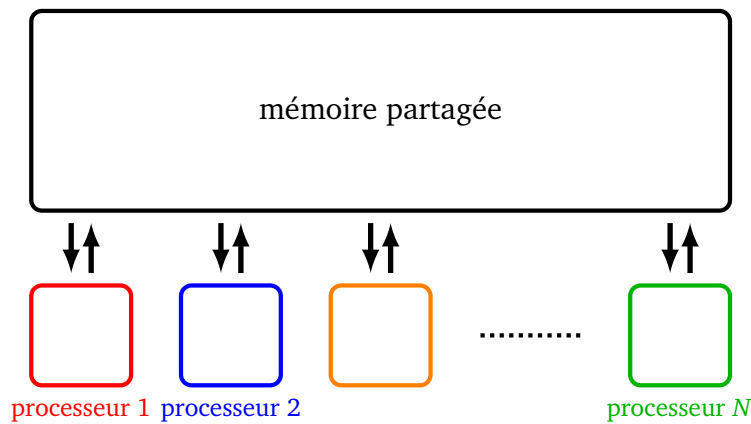
Attention, ce n'est pas toujours possible d'effectuer les calculs en parallèle : par exemple pour calculer $(7 + 5) \times 3 + 2$, il n'y a pas d'autre choix que de calculer séquentiellement, car on a besoin des résultats partiels pour continuer les calculs.

Cours 2 (Calculs en parallèle : un modèle).

Modèle.

Voici le modèle d'ordinateur avec lequel nous travaillerons ici :

- une mémoire partagée qui contient les données, stocke les résultats,
- plusieurs processeurs qui fonctionnent simultanément et ont accès à la mémoire.



Nombre de calculs/temps des calculs.

On souhaite mesurer l'efficacité des algorithmes en parallèle. Un algorithme est divisé en calculs élémentaires. Pour un algorithme séquentiel (avec un seul processeur) chaque tâche est exécutée une par une donc le temps des calculs est égal au nombre de calculs à effectuer. Pour un ordinateur qui effectue des calculs en parallèle, on distingue les deux notions :

- le **nombre de calculs** C est le nombre de tâches élémentaires,
- le **temps des calculs** T est le temps total pour effectuer tous les calculs.

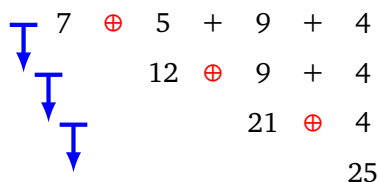
Un calcul correspond à une unité de temps. Deux calculs peuvent être faits successivement pour un temps des calculs $T = 2$, mais si les deux calculs sont effectués en parallèle alors $T = 1$.

Exemple 1.

On veut multiplier tous les éléments d'une liste $(x_0, x_1, \dots, x_{n-1})$ par 2, pour obtenir $(2x_0, 2x_1, \dots, 2x_{n-1})$.

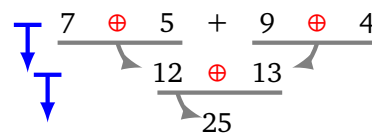
- Si le nombre de processeurs est $N = 1$, alors il faut effectuer successivement chacune des opérations $2x_i$, $i = 0, \dots, n-1$. Le nombre de calculs est $C = n$ et le temps des calculs est $T = n$.
- Si on dispose de $N = 2$ processeurs, alors on peut effectuer les calculs $2x_0$ et $2x_1$ en même temps, puis $2x_2$ et $2x_3$. Au final, le nombre total de calculs reste $C = n$, mais par contre le temps des calculs est divisé par 2 : $T = n/2$ (en supposant n pair, ou bien $T = n/2 + 1$ si n est impair).
- Avec $N = 4$ processeurs, on a toujours $C = n$ et $T \simeq n/4$.
- Si on a beaucoup de processeurs ($N \geq n$) alors $C = n$ et $T = 1$ car tous les calculs peuvent être effectués en même temps.

Exemple 2.



$C = 3 \quad T = 3$

Calculs séquentiels



$C = 3 \quad T = 2$

Calculs en parallèle

Reprenons l'exemple vu plus haut du calcul $7 + 5 + 9 + 4$.

- Calculs séquentiels ($N = 1$). Le nombre d'opérations est $C = 3$: c'est le nombre d'additions (notées $\oplus$ en rouge). Le temps des calculs est $T = 3$ (c'est le nombre d'étapes symbolisées par les flèches verticales bleues).

- Calculs en parallèle avec $N = 2$. Le nombre de calculs est toujours $C = 3$ (il y a autant d'additions $\oplus$ effectuées), par contre le temps des calculs est cette fois $T = 2$ car lors de la première étape deux calculs sont effectués en parallèle.

Activité 1 (Modèle et premiers calculs en parallèle).

Objectifs : simuler le travail d'un ordinateur avec plusieurs processeurs calculant en parallèle et programmer nos premiers algorithmes de calculs en parallèle.

On simule le travail d'un ordinateur effectuant des calculs en parallèle. Notre ordinateur reçoit une liste de n instructions sous la forme d'une chaîne de caractères. L'ordinateur possède N processeurs. Alors :

- le nombre de calculs est $C = n$,
- le temps des calculs est $T = n/N$ (arrondi à l'entier supérieur).

Exemple : si les instructions sont `['2+2', '3*4', '10+1', '8-5']` alors l'ordinateur renvoie `[4, 12, 11, 3]`, le nombre de calculs est toujours $C = n = 4$, le temps des calculs dépend du nombre de processeurs. Par exemple avec $N = 2$, le temps des calculs est $T = 2$, mais si $N = 4$ le temps des calculs est $T = 1$.

1. Notre modèle.

Programme une fonction `calcule_en_parallele(liste_instructions, N)` qui reçoit une liste d'instructions (sous la forme de chaînes de caractères). La fonction renvoie d'abord la liste des résultats mais aussi le nombre de calculs C et le temps des calculs T nécessaires aux N processeurs.

Exemple. Avec les instructions :

`['2+3', '6*7', '8-2', '5+4', '4*3', '12/3']`

et $N = 2$ processeurs la fonction renvoie :

`[5, 42, 6, 9, 12, 4], 6, 3`

Le nombre de calculs est $C = 6$, le temps des calculs est $T = 3$. Teste d'autres valeurs de N .

Indications.

- `eval(chaine)` permet d'évaluer une expression donnée sous la forme d'une chaîne de caractères. Par exemple `eval('8*3')` renvoie 24.
- `ceil(x)` (du module `math`) renvoie la partie entière supérieure d'un nombre. Exemple `ceil(3.5)` vaut 4.

2. Addition de deux vecteurs.

On se donne deux vecteurs $\vec{v}_1 = (x_0, x_1, x_2, \dots, x_{n-1})$ et $\vec{v}_2 = (y_0, y_1, y_2, \dots, y_{n-1})$ (autrement dit deux listes de nombres). Il s'agit de calculer la somme $\vec{v}_1 + \vec{v}_2 = (x_0 + y_0, x_1 + y_1, \dots, x_{n-1} + y_{n-1})$. Programme une fonction `addition_vecteurs(v1, v2)` qui prend en entrée deux listes de nombres et renvoie le vecteur somme $\vec{v}_1 + \vec{v}_2$.

Méthode. Cette fonction doit effectuer les calculs en parallèle en utilisant ta fonction `calcule_en_parallele()`. Il y a un calcul à faire pour chaque composante des vecteurs. En plus cette fonction peut renvoyer le nombre de calculs et le temps des calculs.

Exemple. Avec `v1 = [1, 2, 3, 4]` et `v2 = [10, 11, 12, 13]`, la commande `addition_vecteurs(v1, v2)` renvoie d'une part le résultat `[11, 13, 15, 17]` et si on fixe par exemple le nombre de processeurs à $N = 2$ alors le nombre de calculs est $C = 4$ et le temps des calculs est $T = 2$.

Indications. Il faut transformer chaque calcul en une instruction sous la forme d'une chaîne de caractères. Par exemple la chaîne obtenue par la commande « `str(x) + '*' + str(y)` » sert pour le calcul « $x \times y$ ».

3. Somme des termes.

On se donne un vecteur $\vec{v}$ sous la forme d'une liste de nombres $(x_0, x_1, x_2, \dots, x_{n-1})$. Il s'agit de calculer la somme :

$$S = x_0 + x_1 + x_2 + \dots + x_{n-1} = \sum_{i=0}^{n-1} x_i$$

On propose la méthode suivante :

- on calcule en parallèle la somme de deux termes consécutifs :

$$y_0 = x_0 + x_1 \quad y_1 = x_2 + x_3 \quad y_2 = x_4 + x_5 \quad \dots$$

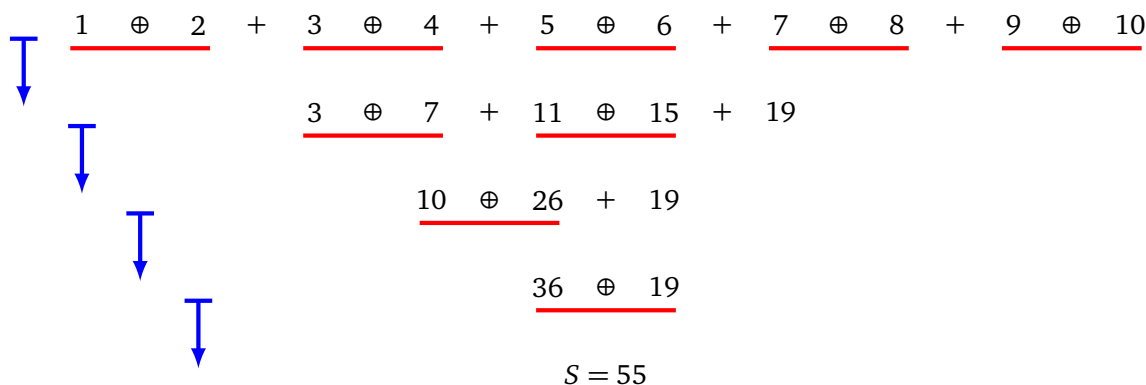
et de façon générale

$$y_i = x_{2i} + x_{2i+1}.$$

On obtient une nouvelle liste $(y_0, y_1, y_2, \dots)$ deux fois plus courte.

- Puis on recommence avec la liste obtenue : $z_0 = y_0 + y_1, z_1 = y_2 + y_3, \dots$ jusqu'à obtenir un seul élément qui est la somme S voulue.

Voici l'algorithme sur l'exemple de la somme $1 + 2 + 3 + \dots + 10$. Les termes sont regroupés par paires (s'il reste un élément isolé à la fin, il est reporté sur la ligne d'après). Le nombre de calculs est $C = 9$ (le nombre d'opérations $\oplus$ soulignées) et si on dispose de $N = 5$ processeurs alors le temps des calculs est $T = 4$ (le nombre de flèches).



Voici le détail de l'algorithme qui en entrée reçoit une liste $\vec{v} = (x_0, x_1, \dots, x_{n-1})$:

- Tant que la longueur n de la liste $\vec{v}$ est strictement plus grande que 1 :
 - Définir une liste $\vec{w}$ dont les composantes y_i sont $y_i = x_{2i} + x_{2i+1}$ pour i variant de 0 à $n/2$.
 - Si n est impair rajouter à $\vec{w}$ le dernier élément de $\vec{v}$.
 - Faire $\vec{v} \leftarrow \vec{w}$.
- À la fin la liste est de longueur 1, l'unique terme donne la somme S cherchée.

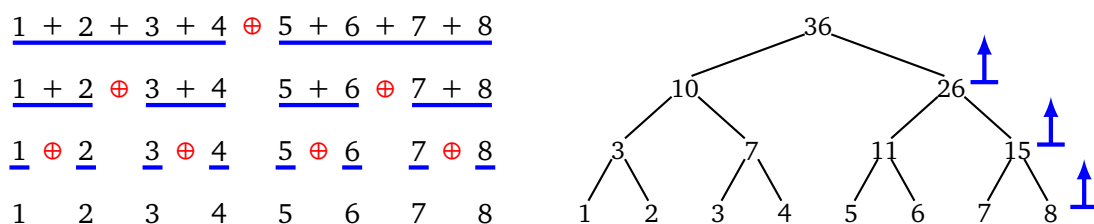
Programme cet algorithme en une fonction `somme(v)`. Cette fonction doit faire appel à la fonction `calcule_en_parallele()` pour le calcul simultané des y_i . Pour cela il faut transformer les opérations $x_{2i} + x_{2i+1}$ en une chaîne de caractères.

En plus de la somme, programme dans un deuxième temps ta fonction de sorte qu'elle renvoie aussi le nombre total de calculs, ainsi que le temps total des calculs (ce sont les sommes de tous les nombres et temps des calculs intermédiaires).

Exemple. Avec $N = 4$ processeurs et $\vec{v} = (1, 2, 3, 4, 5, 6, 7, 8)$ une liste de longueur $n = 8$ alors on trouve une somme de $S = 36$, le nombre de calculs est $C = 7$ (c'est le nombre de signes « + » dans la somme $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$) et un temps des calculs de $T = 3$. Avec $N = 2$ processeurs, le temps des calculs devient $T = 4$.

4. Somme des termes par un algorithme récursif (facultatif).

L'idée est de séparer notre liste en une partie droite et une partie gauche, puis d'itérer le processus. Voici le schéma sur l'exemple de la somme $1 + 2 + 3 + \dots + 8$. Sur la figure de gauche, à lire de haut en bas, les divisions successives en partie droite et partie gauche. Sur la figure de droite l'arbre des calculs, à lire depuis le bas vers le haut.



Algorithme.

- — Entête : `somme_recursive(v)`
- Entrée : $\vec{v} = (x_0, x_1, \dots, x_{n-1})$ une liste de n nombres.
- Sortie : la somme S de ses termes
- Action : fonction récursive.
- **Cas terminaux.**
 - Si $n = 0$, renvoyer 0.
 - Si $n = 1$, renvoyer le seul élément de la liste.
- **Cas général où $n \geq 2$.**
 - Séparer la liste $\vec{v}$ en deux sous-listes (à l'aide du rang $n/2$) une sous-liste des termes de gauche v_gauche et une sous-liste des termes de droite v_droite .
 - Calculer la somme S_g des termes de gauche par l'appel récursif `somme_recursive(v_gauche)`.
 - Calculer la somme S_d des termes de droite par l'appel récursif `somme_recursive(v_droite)`.
 - Renvoyer la somme $S = S_g + S_d$.

Programme cet algorithme en une fonction `somme_recursive(v)`.

Zombies. C'est cette méthode qui s'apparente le plus à la distribution efficace de nos pilules pour contrer les zombies.

Plus difficile. Dans un second temps, modifie ta fonction afin qu'elle renvoie aussi le nombre de calculs et le temps des calculs nécessaires (en supposant qu'il y a suffisamment de processeurs).

5. Produit scalaire.

Programme une fonction `multiplication_vecteurs(v1,v2)` calquée sur le modèle `addition_vecteurs()` qui renvoie le produit terme à terme $(x_0 \times y_0, x_1 \times y_1, \dots, x_{n-1} \times y_{n-1})$ de deux vecteurs $\vec{v}_1 = (x_0, x_1, x_2, \dots, x_{n-1})$ et $\vec{v}_2 = (y_0, y_1, y_2, \dots, y_{n-1})$.

À l'aide de la fonction `somme()` déduis-en une fonction `produit_scalaire(v1,v2)` qui calcule le produit scalaire :

$$\langle \vec{v}_1 | \vec{v}_2 \rangle = x_0 \times y_0 + x_1 \times y_1 + \dots + x_{n-1} \times y_{n-1}$$

Calculs. Le produit scalaire nécessite n multiplications, puis $n-1$ additions, donc un total de $C = 2n-1$ calculs. On prend l'exemple de deux vecteurs de longueur $n = 16$. Avec un seul processeur il faudra un temps des calculs $T = 31$. Combien de temps faut-il si on dispose de $N = 8$ processeurs ?

Vérifie expérimentalement que si n est grand, alors on a $T \simeq C/N$.

I

Activité 2 (Doublons).

Objectifs : retirer les doublons d'une liste à l'aide d'algorithmes adaptés aux calculs en parallèle.

Première méthode : indexation.

Pour cette méthode la liste est une liste d'entiers, par exemple des entiers entre 0 et 99 :

```
liste = [59, 72, 8, 37, 37, 8, 21, 22, 37, 59]
```

On souhaite retirer les doublons, c'est-à-dire obtenir la liste :

```
[59, 72, 8, 37, 21, 22]
```

L'idée est de remplir petit à petit une grande table d'indexation :

- au départ la table contient 100 zéros (un pour chaque entier possible entre 0 et 99),
- pour chaque élément i de la liste initiale on regarde la table au rang i :
 - s'il y a un 0 c'est que l'élément est nouveau, on le conserve dans une nouvelle liste et on place un 1 au rang i de la table,
 - s'il y a déjà un 1 dans la table c'est que l'élément i a déjà été indexé, on ne le conserve pas dans la nouvelle liste.

Programme cet algorithme en une fonction `enlever_tous_doublons(liste)`.

Exemple. Prenons un exemple plus simple avec des entiers de 0 à 9 et la liste :

```
liste = [3, 5, 4, 5, 8, 8, 4]
```

La table d'indexation au départ ne contient que des 0 :

```
table = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Puis on parcourt la liste :

- le premier élément est 3 : on place un 1 au rang 3, la table d'indexation est maintenant [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] (la numérotation commence au rang 0).
- les éléments suivants sont 5 puis 4, on place des 1 au rang 5 puis au rang 4, la table est maintenant [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
- ensuite on retrouve l'élément 5, on sait qu'on a déjà pris en compte cet élément car au rang 5 de la table il y a un 1, on ne fait donc rien,
- ensuite l'élément est 8, la table devient [0, 0, 0, 1, 1, 1, 0, 0, 1, 0], puis on retrouve 8 qui ne change rien,
- enfin on trouve 4 que l'on a déjà rencontré et qui ne change pas la table.

On ne retient que les éléments lors de leur première apparition, la liste sans doublons est donc :

```
[3, 5, 4, 8]
```

Calculs en parallèle. Cet algorithme est facile à implémenter en parallèle : la table est située dans la mémoire globale et le parcours des éléments se fait en parallèle, s'il y a un 0 dans la table on conserve l'élément, s'il y a un déjà 1 on l'oublie.

Inconvénients. La méthode présente deux gros inconvénients : d'une part elle n'est valable que pour des listes d'entiers et surtout il faut construire une table qui peut être immense comparée à la liste initiale, par exemple même pour une petite liste d'entiers entre 0 et 999 999 il faut commencer par construire une table de longueur un million. La seconde méthode va remédier à ces deux problèmes.

Seconde méthode : table de hachage.

On souhaite enlever les doublons de la liste :

```
['LAPIN', 'CHAT', 'ZEBRE', 'CHAT', 'CHIEN',  
'TORTUE', 'CHIEN', 'SINGE', 'SINGE', 'CHAT']
```

1. **Fonction de hachage.** À une chaîne de caractères et un entier p , on va associer un entier h avec $0 \leq h < p$; h sera appelé le *hachage* du mot modulo p . La méthode est la suivante :

- on attribue à chaque lettre une valeur : **A** vaut 0, **B** vaut 1, ..., **Z** vaut 25,
- on prend la valeur de la lettre de rang 0 dans le mot, on multiplie par 26^0 ,
- auquel on ajoute la valeur de la lettre de rang 1, multipliée par 26^1 ,
- ...
- auquel on ajoute la valeur de la lettre de rang k , multipliée par 26^k ,
- ...
- de plus les calculs se font modulo l'entier p donné, donc la somme totale est un entier h avec $0 \leq h < p$.

Programme une fonction `hachage(mot, p)` qui renvoie le hachage du mot donné modulo p .

Exemples. Voici le détail des calculs du hachage de **LAPIN** modulo $p = 10$:

lettre	L	A	P	I	N
valeur	11	0	15	8	13
facteur	26^0	26^1	26^2	26^3	26^4
produit	11	0	10 140	140 608	5 940 688
modulo $p = 10$	1	0	0	8	8

Donc le hachage de **LAPIN** modulo 10 est $h = 1 + 0 + 0 + 8 + 8 \pmod{10} = 17 \pmod{10} = 7$.

Autre exemple avec **CHIEN** :

lettre	C	H	I	E	N
valeur	2	8	8	4	13
facteur	26^0	26^1	26^2	26^3	26^4
produit	2	182	5 408	70 304	5 940 688
modulo $p = 10$	2	2	8	4	8

Donc le hachage de **CHIEN** modulo 10 est $h = 2 + 2 + 8 + 4 + 8 \pmod{10} = 24 \pmod{10} = 4$.

Vérifie que le hachage de **SINGE** modulo 10 vaut aussi $h = 4$.

Important. Il peut donc y avoir deux mots différents qui ont la même valeur de hachage.

Indications. Pour récupérer le rang d'une lettre, définis une chaîne contenant toutes les lettres :

ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

Puis utilise la méthode `index()`, pour déterminer le rang d'un caractère stocké dans la variable `c` :

ALPHABET.index(c)

Par exemple `ALPHABET.index('D')` renvoie 3 (la numérotation commence à 0).

2. **Table de hachage et élimination de certains doublons.** Maintenant que l'on a associé un entier à chaque élément de la liste alors le principe est assez similaire à la première méthode, mais avec une table (dite table de hachage) beaucoup plus petite.

Voici l'algorithme :

- on fixe un entier p ,
- au départ on définit une table `table` qui contient p chaînes vides '' (une pour chaque valeur de hachage possible entre 0 et $p - 1$),
- pour chaque mot de la liste initiale on calcule son hachage h (qui dépend du mot et de p) et on regarde `table[h]`, la valeur de la table au rang h :
 - s'il y a une chaîne vide '' c'est que le mot est nouveau, on le conserve dans une nouvelle liste et en plus on place ce mot au rang h de la table : `table[h] = mot`;

- s'il y a déjà une chaîne non-vide :
 - soit c'est la même chaîne que mot, auquel cas mot est un doublon que l'on ne conserve pas,
 - soit c'est une chaîne différente, auquel cas on conserve le mot.

Programme cet algorithme en une fonction `enlever_des_doublons(liste, p)` qui renvoie la liste originale des mots sans les doublons détectés par la fonction de hachage modulo p .

Prenons l'exemple de la liste `['CHIEN', 'LAPIN', 'CHIEN', 'SINGE', 'SINGE']` avec les calculs modulo $p = 10$.

- Au départ la table de hachage contient 10 chaînes vides :


```
['', '', '', '', '', '', '', '', '', '']
```
- La hachage de **CHIEN** modulo 10 vaut $h = 4$ (voir la question précédente), donc on place ce mot dans la table au rang 4, la table est maintenant :


```
table : ['', '', '', '', 'CHIEN', '', '', '', '', '']
```

 et la nouvelle liste : `['CHIEN']`
- La hachage de **LAPIN** modulo 10 vaut $h = 7$, donc on place ce mot dans la table au rang 7, la table est maintenant :


```
table : ['', '', '', '', 'CHIEN', '', '', 'LAPIN', '', '']
```

 nouvelle liste : `['CHIEN', 'LAPIN']`
- Le mot suivant est encore **CHIEN**, de hachage $h = 7$. Il y a déjà un mot au rang 7, et comme c'est déjà **CHIEN** alors on ne change pas la table et on ne retient pas ce mot (rien ne change) :


```
table : ['', '', '', '', 'CHIEN', '', '', 'LAPIN', '', '']
```

 nouvelle liste : `['CHIEN', 'LAPIN']`
- Le mot suivant est **SINGE**, de hachage $h = 4$. Il y a déjà un mot au rang 4, mais c'est le mot **CHIEN**, comme ces deux mots sont différents alors on ne change pas la table mais on conserve le mot **SINGE** :


```
table : ['', '', '', '', 'CHIEN', '', '', 'LAPIN', '', '']
```

 nouvelle liste : `['CHIEN', 'LAPIN', 'SINGE']`
- Le mot suivant est encore **SINGE**, de hachage $h = 4$. Pour la même raison on conserve encore le mot **SINGE** :


```
table : ['', '', '', '', 'CHIEN', '', '', 'LAPIN', '', '']
```

 nouvelle liste : `['CHIEN', 'LAPIN', 'SINGE', 'SINGE']`

Notre méthode a donc supprimé un doublon de **CHIEN**, mais pas le doublon de **SINGE**. L'explication est simple : deux mots identiques ont bien la même valeur de hachage, mais il se peut que cela se produise aussi pour deux mots différents ! En choisissant p assez grand, ces accidents deviennent assez rares.

Par exemple ici **CHIEN** et **SINGE** ont la même valeur de hachage $h = 4$ pour $p = 10$, mais si on fixe $p = 11$ alors le hachage de **CHIEN** vaut $h = 2$ et celui de **SINGE** vaut $h = 5$.

3. Itération et élimination de tous les doublons.

Programme une fonction `iterer_enlever_des_doublons(liste, nb_iter)` qui itère la fonction précédente avec différentes valeurs de p :

- commence avec p qui vaut deux fois la longueur de la liste,
- applique la fonction précédente avec cette valeur de p ,
- à partir de la liste renvoyée, retire les doublons avec cette fois $p \leftarrow p + 1$,
- itère en incrémentant p à chaque fois.

En général, avec trois itérations il y a peu de chance qu'il reste des doublons !

Exemple. Reprenons la liste :

`['CHIEN', 'LAPIN', 'CHIEN', 'SINGE', 'SINGE']`

Il y a 5 mots, donc on fixe $p = 10$. On retire d'abord les doublons modulo $p = 10$, on obtient la liste :

`['CHIEN', 'LAPIN', 'SINGE', 'SINGE']`

On repart de cette liste et on retire maintenant les doublons modulo $p = 11$, on obtient une liste sans doublons :

`['CHIEN', 'LAPIN', 'SINGE']`

Encore une fois, il serait facile d'effectuer le travail en parallèle sur les éléments : calcul du hachage et test s'il est présent dans la table.

Activité 3 (Calculs en parallèle sur les listes).

Objectifs : voir des algorithmes bien adaptés aux calculs en parallèle pour les listes. Les fonctions de cette activités sont des fonctions récursives.

1. Maximum d'une liste.

Il est facile de trouver le maximum d'une liste en parcourant la liste du début à la fin. On suppose ici que l'on a deux processeurs (ou plus). Voici l'idée pour profiter des calculs en parallèle : on divise la liste en deux, on cherche le maximum de la partie gauche (avec des processeurs), on cherche le maximum de la partie droite (avec d'autres processeurs) et on compare les deux maximums pour renvoyer le résultat.

Voici l'algorithme qui définit une fonction récursive `maximum()` renvoyant le maximum d'une liste de nombre.

Algorithme.

- — Entête : `maximum(liste)`
 - Entrée : `liste`, une liste de n nombres.
 - Sortie : le maximum de la liste.
 - Action : fonction récursive.
- **Cas terminaux.**
 - Si $n = 0$, renvoyer une très grande valeur négative (par exemple -1000).
 - Si $n = 1$, renvoyer le seul élément de la liste.
- **Cas général où $n \geq 2$.**
 - Diviser la liste en deux parties : une partie gauche `liste_gauche` et une partie droite `liste_droite`.
 - Calculer le maximum de la partie gauche par un appel récursif `maximum(liste_gauche)`.
 - Calculer le maximum de la partie droite par un appel récursif `maximum(liste_droite)`.
 - Renvoyer le plus grand de ces deux maximums (à l'aide de la fonction habituelle `max(a, b)`).

L'infini. La valeur -1000 du cas terminal est une valeur qui doit être plus petite que toutes les valeurs de la liste. Il n'est donc pas sûr que -1000 soit suffisant. La bonne solution est d'utiliser la valeur infinie `inf` (qui correspond à $+\infty$) disponible depuis le module `math`. Par exemple « $3 < \text{inf}$ » vaut « Vrai ». De même pour $-\infty$, « $3 > -\text{inf}$ » vaut « Vrai ».

2. Termes pairs d'une liste.

Il s'agit de ne conserver que les termes pairs d'une liste d'entiers. Par exemple à partir de la liste `[7, 2, 8, 12, 5, 8]`, on ne conserve que `[2, 8, 12, 8]`. L'idée pour profiter des calculs en parallèle est

similaire à celle de la question précédente : on divise la liste en deux, on cherche les termes pairs de la partie gauche, on cherche les termes pairs de la partie droite et on concatène ces deux listes.

Écris en détails l'algorithme récursif correspondant à cette idée. Réfléchis-bien aux cas terminaux (si la liste est vide bien sûr, on renvoie la liste vide, si la liste ne contient qu'un élément, que renvoie-t-on ?).

Programme ton algorithme en une fonction `extraire_pairs(liste)`.

3. Premier rang non nul.

On considère une liste qui contient beaucoup de 0. Il s'agit de trouver le rang du premier terme non nul. Par exemple pour la liste $[0, 0, 0, 0, 0, 1, 0, 1, 1, 0]$ le rang du premier terme non nul est 5 (on commence à compter à partir du rang 0).

Voici l'idée pour profiter des calculs en parallèle :

- on sépare la liste en une partie gauche et une partie droite,
- par un appel récursif sur la partie gauche, on sait s'il y a un élément non nul dans la partie gauche, si c'est le cas on renvoie le rang et c'est fini, si ce n'est pas le cas on passe à la suite avec la partie droite,
- si l'étude de la partie gauche n'a rien donné, on effectue un appel récursif sur la partie droite, s'il y a un élément non nul dans la partie droite, on renvoie le rang augmenté de la longueur de la liste de gauche et c'est fini.

Écris en détails l'algorithme et programme une fonction `premier_rang(liste)`.

Indications.

- La fonction renvoie `None` si tous les éléments sont nuls.
- Cas terminaux : si la liste est vide, on renvoie `None` ; si la liste ne contient qu'un élément, on renvoie `None` si cet élément est nul, et le rang 0 sinon.
- Si l'étude de la sous-liste de gauche n'a rien donné il ne faut pas oublier de décaler le rang du premier terme non nul de la sous-liste de droite.
- Par exemple pour la liste $[0, 0, 0, 0, 0, 1, 0, 1, 1, 0]$, la partie gauche $[0, 0, 0, 0, 0]$ (de longueur 5) a tous ses termes nuls, pour la partie droite $[1, 0, 1, 1, 0]$ le premier terme non nul est de rang 0, mais dans la liste de départ ce terme est de rang $5 + 0 = 5$.

Cours 3 (Sommes partielles).

Soit une suite d'éléments :

$$[x_0, x_1, x_2, \dots, x_{n-1}],$$

la liste des **sommes partielles** est :

$$[x_0, x_0 + x_1, x_0 + x_1 + x_2, \dots, x_0 + x_1 + x_2 + \dots + x_{n-1}]$$

Exemples. La liste $[1, 2, 3, 4, 5, 6, 7, 8]$ a pour sommes partielles

$$[1, 3, 6, 10, 15, 21, 28, 36].$$

Autre exemple, la liste $[10, 4, 0, 2, 1, 0, 3, 21]$ a pour sommes partielles $[10, 14, 14, 16, 17, 17, 20, 41]$.

La k -ème somme partielle est donc :

$$S_k = x_0 + x_1 + x_2 + \dots + x_k$$

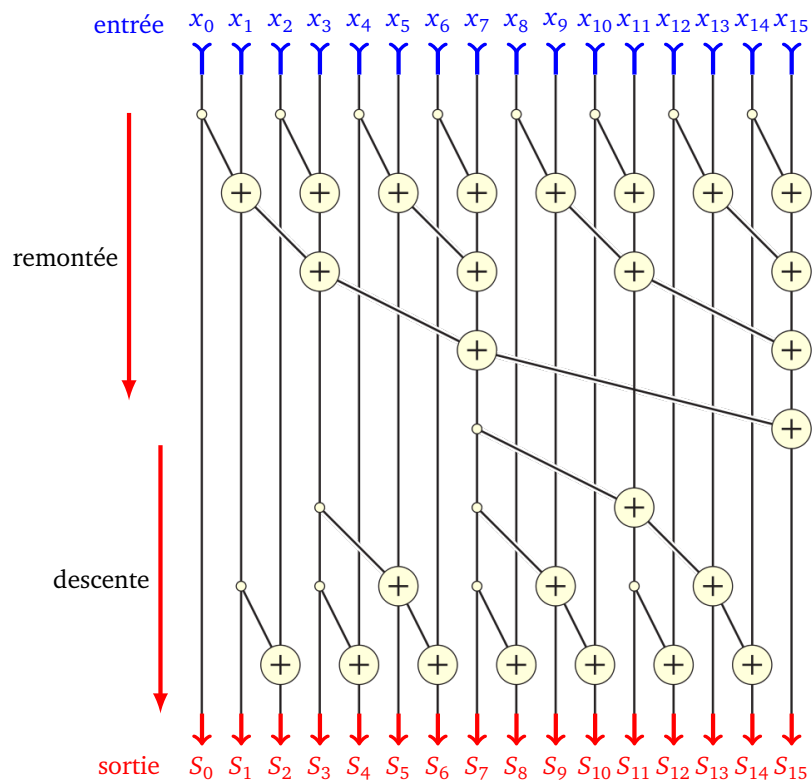
et s'obtient à partir de la précédente par la formule

$$S_k = S_{k-1} + x_k$$

en initialisant $S_0 = x_0$ (ou mieux $S_{-1} = 0$).

Calculs en parallèle.

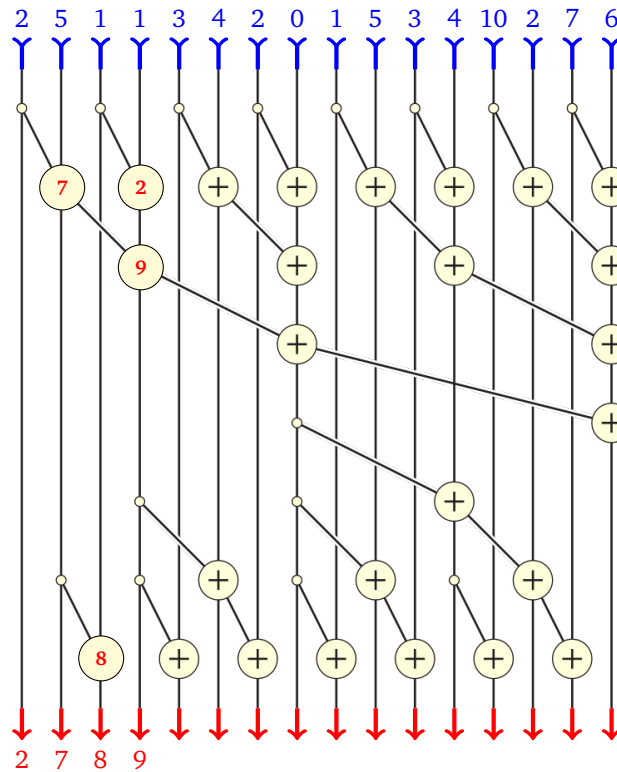
Il existe une méthode astucieuse pour effectuer les calculs en parallèle. Le principe est expliqué sur la figure ci-dessous.



Le schéma se lit de haut en bas. En haut il y a les entrées, ici une liste de 16 éléments. En bas la liste des sommes partielles. Chaque nœud $\oplus$ signifie l'addition des deux nombres issus des arêtes au-dessus. Tout ce qui sort vers le bas du nœud est le résultat de cette addition.



Exemple. Voici le début des calculs, à toi de les finir !



Activité 4 (Sommes partielles).

Objectifs : il est très facile de calculer les sommes partielles par un algorithme séquentiel, cependant il existe un algorithme récursif qui permet de faire ces calculs en parallèle.

1. **Algorithme séquentiel.** Programme une fonction `sommes_partielles(liste)` qui renvoie la liste des sommes partielles après avoir parcouru (une seule fois!) la liste élément par élément.
2. **Algorithme parallèle.** Implémente l'algorithme suivant (qui correspond aux explications données ci-dessus) en une fonction récursive `sommes_partielles_recuratif(liste)`. On suppose que la longueur de la liste est une puissance de 2.

Algorithme.

- Entête : `sommes_partielles_recuratif(liste)`
- Entrée : `liste = [x0, x1, ..., xn-1]` une liste de n nombres (n est une puissance de 2).
- Sortie : la liste de ses sommes partielles.
- Action : fonction récursive.

• **Cas terminal.**

Si $n = 1$, renvoyer la liste (qui ne contient qu'un élément).

• **Cas général où $n \geq 2$.****Remontée.**

- Former la liste `sous_liste` de taille $n/2$ constituée de l'addition d'un terme de rang pair et d'un terme de rang impair :

$$[x_0 + x_1, x_2 + x_3, \dots]$$

- Avec cette `sous_liste`, faire un appel récursif :

`sommes_partielles_recuratif(sous_liste)`

et nommer le résultat `liste_remontee` qui est une liste $[y_0, y_1, \dots, y_{n/2-1}]$.

Descente.

- Initialiser une liste `liste_descente` qui contient seulement x_0 .
- Pour i allant de 1 à $n-1$:
- Si i est pair, ajouter à `liste_descente` l'élément

$$y_{i/2-1} + x_i,$$

- sinon, ajouter à `liste_descente` l'élément

$$y_{(i-1)/2}.$$

- Renvoyer la liste `liste_descente`.

3. Applications : sélection des éléments par un filtre.

Imaginons une liste quelconque, par exemple :

`liste = [15, 18, 16, 11, 15, 19, 13, 12]`

dont on souhaite ne conserver que certains éléments. Pour cela on définit un filtre :

`filtre = [0, 0, 1, 0, 1, 0, 0, 1]`

On ne retient que ceux qui sont en correspondance avec un 1 :

`selec = [16, 15, 12]`

C'est très facile à programmer de façon séquentielle (fais-le) et même en parallèle si on ne se préoccupe pas de l'ordre des éléments (on autorise la sortie $[15, 12, 16]$ par exemple). Voici un algorithme pour le faire en parallèle en préservant l'ordre des éléments.

- On calcule la liste des sommes partielles du filtre. Sur notre exemple, cela donne :

`sommes = [0, 0, 1, 1, 2, 2, 2, 3]`

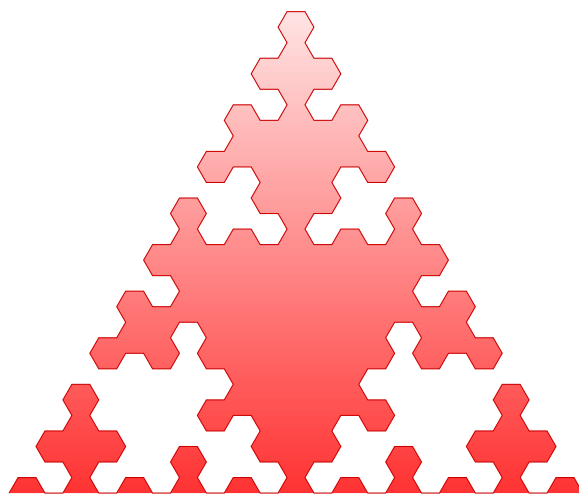
Le dernier élément n (ici $n = 3$) donne la taille de la liste finale `selec`. On initialise donc une liste `selec` de taille n .

- Les sommes partielles donnent le rang des éléments à conserver dans la liste finale (avec un décalage de 1). Comment ? Par exemple on sait qu'il faut conserver l'élément de rang 2 de la liste initiale, en effet pour $i = 2$ on a `filtre[i] = 1` (et pas 0), donc il faut garder l'élément `liste[i]` qui vaut 16. La somme partielle au rang 2, `sommes[i]`, vaut 1, alors 16 aura comme rang final 0 (il y a un décalage de 1). Pour 15, sa somme partielle associée est 2, il sera donc au rang 1 ; pour 12, sa somme partielle est 3 il sera donc au rang 2.

- Ainsi pour i indexant les éléments de la liste initiale, si on doit conserver l'élément de rang i , alors il sera au rang $\text{sommes}[i] - 1$ dans la liste finale. Autrement dit :
$$\text{selec}[\text{sommes}[i] - 1] = \text{liste}[i]$$
- Les éléments à conserver sont maintenant dans la liste `selec`.

Programme cet algorithme en une fonction `selection(liste, filtre)`. (Si tu utilises la fonction de la question précédente, les longueurs des listes doivent être des puissances de 2.)

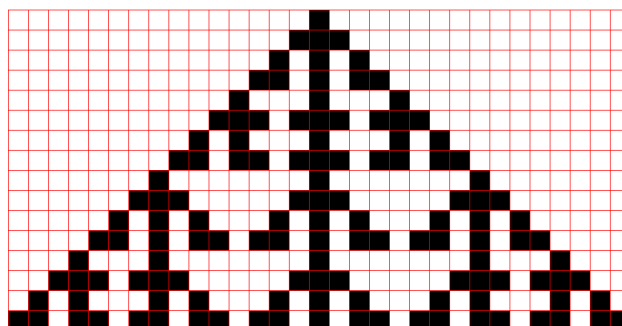
TROISIÈME PARTIE



PROJETS

Automates

Tu vas programmer des automates cellulaires qui, à partir de règles simples, produisent des visualisations amusantes.



Activité 1 (Une suite logique).

Objectifs : programmer une suite logique amusante (mais pas nécessaire pour l'activité suivante).

Voici une suite :

```

1
11
21
1211
111221
312211
13112221
1113213211

```

Pour passer d'une ligne à la suivante, il suffit de lire à haute voix en comptant les nombres ! Par exemple la ligne 1211 est lue « un un (pour 1), un deux (pour 2), deux un (pour 1 1) », la ligne d'après est donc 111221 ! Cette dernière ligne se lit « trois uns, deux deux, un un » donc la ligne suivante sera 312211. Programme une fonction `lecture(mot)` qui calcule et renvoie la lecture de la chaîne `mot`. Par exemple `lecture("1211")` renvoie "111221".

- Essaie de programmer cette fonction sans lire les indications suivantes !
- *Indications.* Tu peux utiliser trois variables : une variable qui lit chaque caractère du mot, une variable correspondant au caractère précédent, un compteur à incrémenter si ces deux caractères sont égaux.
- *Algorithme.* Si tu n'y arrives pas tout seul, voici les grandes lignes d'un algorithme possible.

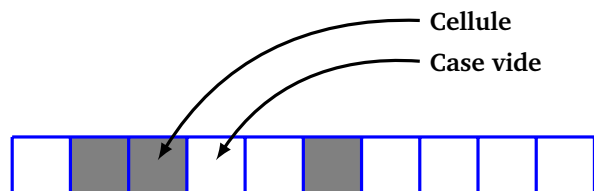
Pour chaque caractère du mot :

- si le caractère est le même que le caractère précédent, incrémenter le compteur,
- sinon, rajouter au mot à créer la valeur du compteur suivie du caractère précédent.

À la fin, il faut aussi rajouter au mot à créer la valeur du compteur suivie du caractère précédent.
Question. Trouve le premier mot qui contient 33.

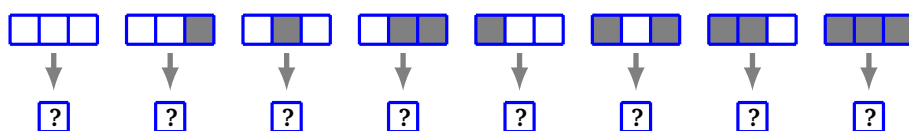
Cours 1 (Automates linéaires).

On travaille sur des lignes superposées, formées de cases. Chaque case peut contenir une cellule (la case est alors noire/contient 1) ou être vide (la case est blanche/contient 0).



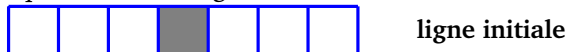
Un **automate linéaire** est une règle qui à partir du contenu de trois cases consécutives sur une ligne, détermine le contenu de la case sur la ligne du dessous.

La **règle** est donc donnée par la liste des 8 configurations possibles au départ, avec pour chacune la naissance ou pas d'une cellule en-dessous.

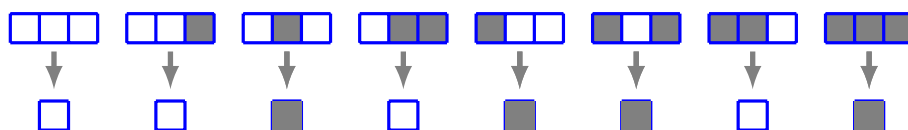


Exemple.

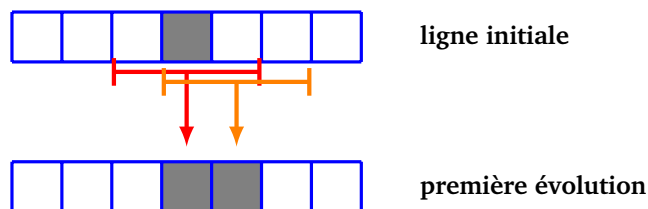
- Partons d'une seule cellule qui sera sur la ligne du haut.



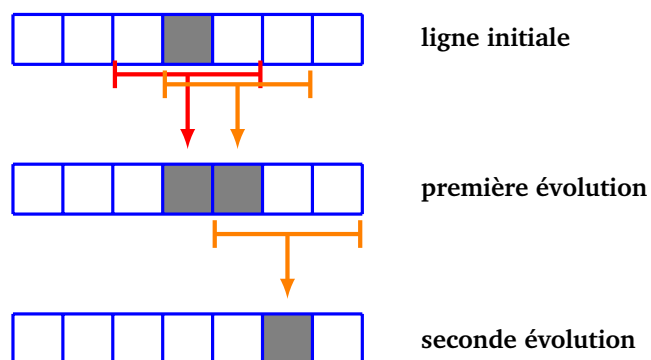
- Et choisissons la règle définie par les configurations :



- Pour décider de la naissance d'une cellule dans une case de la ligne en-dessous, on regarde les trois cases au-dessus et on applique la règle. Sur le dessin ci-dessous deux cellules sont vivantes après l'évolution (on a indiqué par des flèches seulement les règles pour lesquelles une cellule apparaît).

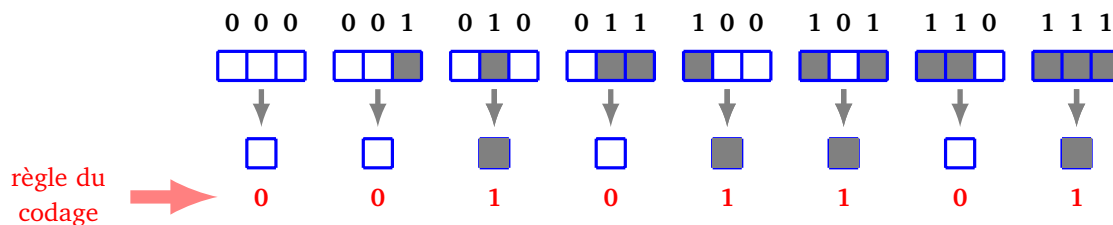


- On peut itérer le processus. Une seule cellule apparaît lors de cette seconde évolution.



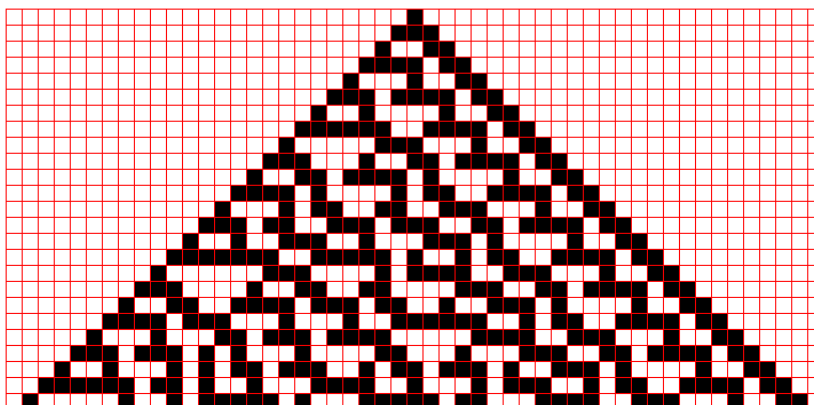
Notations.

- On note 0 pour une case vide et 1 pour une case contenant une cellule vivante.
- Une ligne est représentée par une liste de 0 et de 1. Par exemple $[0, 0, 0, 1, 0, 1, 0, 1, 0, 0]$ est une ligne de 10 cases, contenant 3 cellules.
- La règle est codée par une liste de 0 et 1 et de longueur 8, déterminée par l'image des 8 configurations possibles. Par exemple : $[0, 0, 1, 0, 1, 1, 0, 1]$ correspond à la règle :

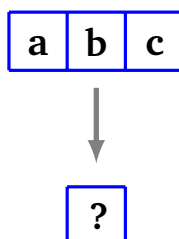


Activité 2 (Automates linéaires).

Objectifs : calculer et afficher les automates linéaires.



1. **Évolution d'une cellule.** Programme une fonction `cellule_suivante(a, b, c, regle)` qui calcule et renvoie la couleur (0 ou 1) de la case située sous les trois cases contenant a, b, c selon la règle donnée.



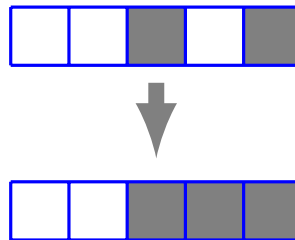
- a, b, c sont les couleurs (0 ou 1) des cases situées au-dessus (a est la case la plus à gauche).
- La loi de transformation est donnée par la liste `regle` formée d'une suite de 8 entiers 0 ou 1.
- Exemple avec `regle = [0,0,1,0,1,1,0,1]` alors `cellule_suivante(0,0,0, regle)` renvoie 0, `cellule_suivante(0,0,1, regle)` renvoie aussi 0, `cellule_suivante(0,1,0, regle)` renvoie 1, etc.
- Si tu ne veux pas écrire les 8 cas possibles, calcule $4a + 2b + c$!

2. **Affichage de la règle.** Déduis-en une fonction `affiche_regle(regle)` qui affiche à l'écran la règle donnée sous la forme « $a, b, c \rightarrow d$ » où d est la couleur de la nouvelle case, par exemple :

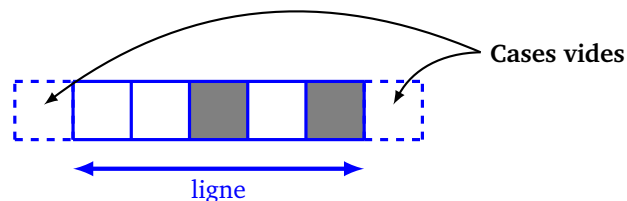
```
0 0 0 -> 0
0 0 1 -> 0
0 1 0 -> 1
...
```

3. **Évolution d'une ligne.** Programme une fonction `ligne_suivante(ligne, regle)` qui à partir d'une liste `ligne` formée de 0 et 1, calcule les cellules de la ligne suivante (renvoyée sous la forme d'une liste de 0 et de 1).

Exemple. Avec la règle `[0,0,1,0,1,1,0,1]` alors la ligne suivant `[0,0,1,0,1]` est la ligne `[0,0,1,1,1]`.



Remarque. Lors du calcul des cases situées à l'une des deux extrémités de la ligne, on considère qu'au delà, il n'y a pas de cellule (c'est donc comme si à droite et à gauche de la ligne initiale il y avait un 0).

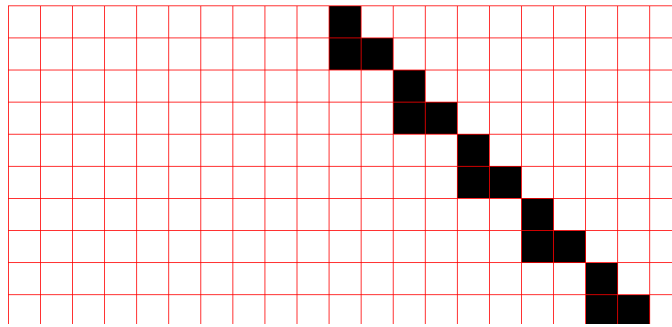


4. **Itérations.** Déduis-en une fonction `plusieurs_lignes(n, ligne, regle)` qui affiche sur le terminal les n lignes qui suivent la ligne donnée.

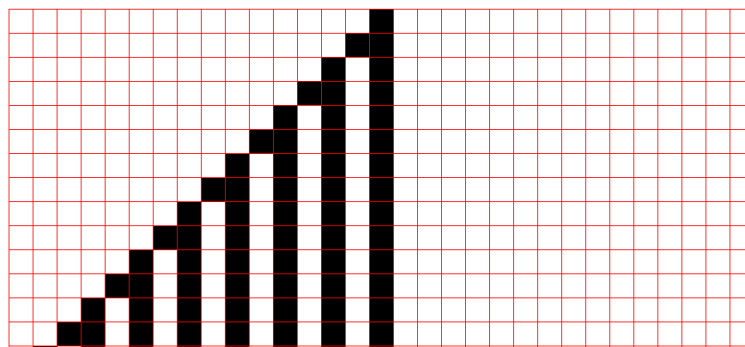
Exemple. Toujours avec la règle `[0,0,1,0,1,1,0,1]`, pars d'une ligne définie par une seule cellule au milieu :

$$\text{ligne} = [0]*10 + [1] + [0]*10$$

Alors l'itération du processus correspond à l'évolution d'une cellule, qui voyage vers la droite en se dédoublant une fois sur deux.



5. **Affichage.** Programme une fonction `afficher_lignes(n, ligne, regle)` qui réalise un bel affichage graphique d'une ligne de cellules et de son évolution sur n lignes. La présence d'une cellule (marquée par 1) est affichée par une case noire, l'absence de cellule (marquée par 0) est affichée par une case blanche.



6. **Numérotation des règles.** Il y a en tout $2^8 = 256$ règles possibles, car une règle est une liste de 8 bits. On décide donc de numéroter la règle en fonction du nombre binaire représenté par la liste :

$$\underbrace{[0, 0, 1, 0, 1, 1, 0, 1]}_{\text{règle}} \longleftrightarrow \underbrace{0.0.1.0.1.1.0.1}_{\text{nombre binaire}} \longleftrightarrow \underbrace{45}_{\text{numéro}}$$

Écris une fonction `definir_regle(numero)` qui n'est autre que la conversion d'un entier en écriture binaire sur 8 bits. Par exemple `definir_regle(45)` renvoie la règle `[0, 0, 1, 0, 1, 1, 0, 1]`.

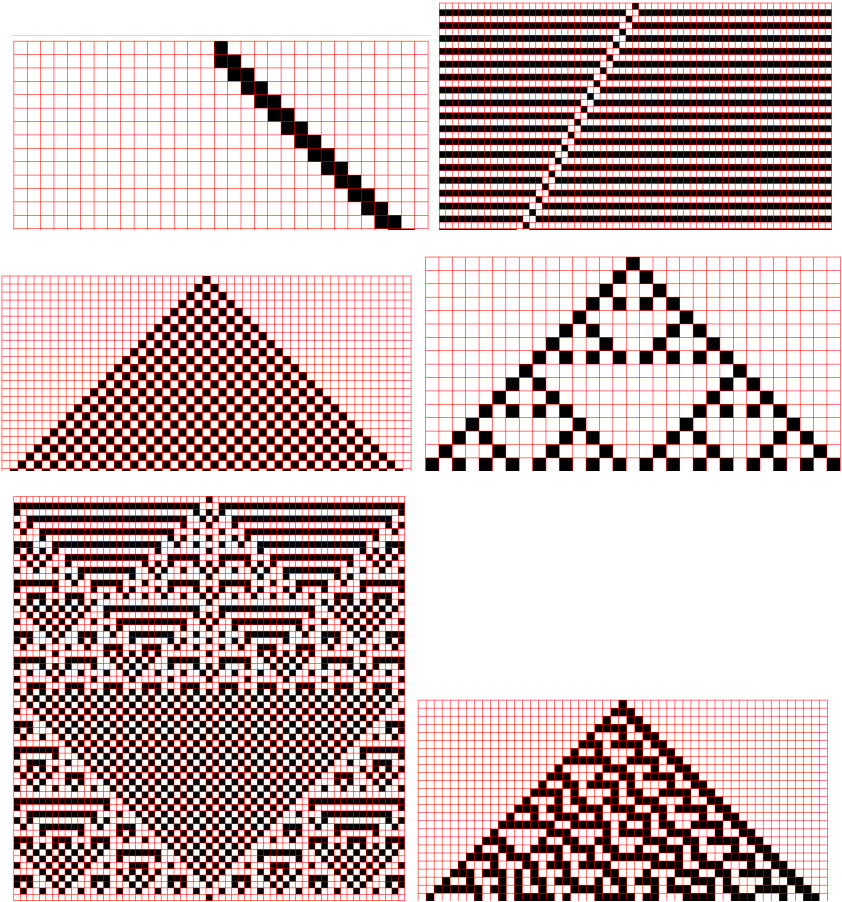
Algorithme.

- — Entrée : un entier n entre 0 et 255.
- Sortie : le nombre n en écriture binaire sous la forme d'une liste de 8 bits.
- Démarrer avec une liste vide.
- Répéter 8 fois :
 - Ajouter $n \% 2$ au début de la liste.
 - Faire $n \leftarrow n // 2$.
- Renvoyer la liste.

7. Types d'automates.

En partant d'une seule cellule, essaie de trouver différents types de comportements :

- des automates cellulaires qui convergent vers un état stable (voire vide),
- des automates cellulaires qui convergent vers un état périodique,
- des automates cellulaires ayant des structures symétriques (par exemple, qui réalisent des triangles de Sierpinski),
- des automates cellulaires avec des structures qui semblent aléatoires.



Cryptographie

Tu vas jouer le rôle d'un espion qui intercepte des messages secrets et tente de les décrypter.

Cours 1 (Chiffre de César).

Le **chiffre de César** est une façon simple de coder un message afin de conserver le secret du contenu jusqu'à son destinataire. Il s'agit tout simplement de décaler chaque lettre du message. Voyons l'exemple d'un décalage de trois lettres : **A** devient **D** ; **B** devient **E** ; etc.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

lettres du message en clair

lettres du message codé

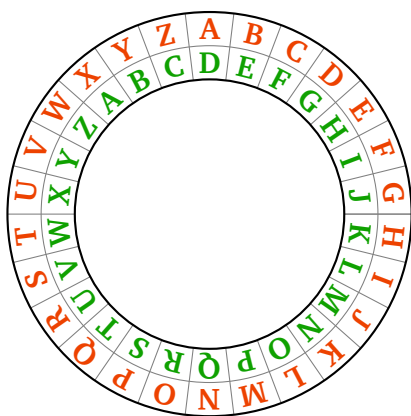
Par exemple le message :

C A P T U R E Z I D E F I X

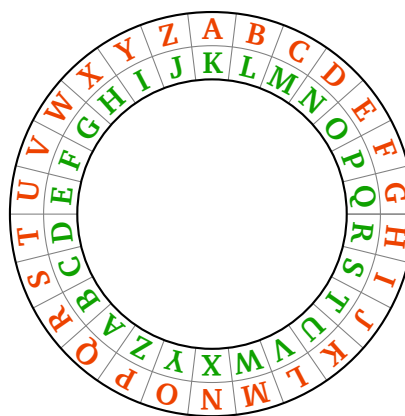
se chiffre en :

F D S W X U H C L G H I L A

Une autre façon de présenter le décalage est de placer les alphabets en clair (anneau extérieur en rouge) et codé (anneau intérieur en vert) sur deux roues concentriques. À gauche un décalage avec $k = 3$ et à droite un décalage avec $k = 10$.



Décalage $k = 3$



Décalage $k = 10$

Pour chiffrer des messages tu passes des lettres rouges à l'extérieur aux lettres vertes à l'intérieur. Pour déchiffrer des messages il suffit de faire l'opération inverse : passer des lettres vertes aux lettres rouges !

Déchiffre à la main les messages suivants :

- **E O R T X H C D V W H U L A** chiffré avec un décalage $k = 3$.
- **Y E O C D Z K X Y B K W S H** chiffré avec un décalage $k = 10$.

Cours 2 (Passer d'un caractère à un nombre et inversement).

On préfère travailler avec des nombres qu'avec des lettres !

- **Alphabet.** Tu vas définir une constante globale `Alphabet` qui contient toutes les lettres de l'alphabet :

`Alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"`

- **Numérotation.** On numérote chaque lettre : 0 pour A, 1 pour B, 2 pour C, ..., 25 pour Z.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

- **D'un numéro vers une lettre.** Pour récupérer la lettre correspondant au rang i , il suffit d'écrire :

`Alphabet[i]`

Par exemple `Alphabet[2]` vaut 'C'.

- **D'une lettre vers son numéro.** Pour récupérer le rang d'une lettre, notée `car`, il suffit d'écrire :

`Alphabet.index(car)`

Par exemple `Alphabet.index('D')` vaut 3.

- **César.** Le chiffrement de César de décalage k correspond juste à une addition : le numéro j du caractère chiffré est égal au numéro i du caractère en clair auquel on ajoute le décalage k , le tout modulo 26 :

$$j = (i + k) \% 26$$

Par exemple avec un décalage $k = 3$:

- le caractère numéro 0 (A) est chiffré en le caractère numéro $0 + 3 = 3$ (D),
- le caractère numéro 1 (B) est chiffré en le caractère numéro $1 + 3 = 4$ (E),
- ...
- le caractère numéro 25 (Z) est chiffré en le caractère numéro $25 + 3 = 28$ qui vaut 2 modulo 26 (c'est donc bien C).

Activité 1 (Chiffre de César).

Objectifs : programmer le chiffrement et le déchiffrement du chiffre de César.

1. Programme une fonction `chiffre_cesar_caractere(car, k)` qui renvoie le caractère de l'alphabet situé k rang après (modulo 26).

Indications.

- Récupère le rang i dans `Alphabet` du caractère à chiffrer.
- Ajoute k modulo 26 par la formule : $j = (i + k) \% 26$.
- Renvoie le caractère correspondant au rang j de `Alphabet`.

Exemple. Avec un décalage de $k = 3$, A devient D et Z devient C.

2. Programme une fonction `chiffre_cesar_phrase(phrase, k)` qui renvoie la phrase codée par un décalage de César k .

Indication. Si un caractère n'est pas une lettre majuscule alors conserve-le tel quel dans la phrase chiffrée. Le test se fait par :

`if car not in Alphabet:`

Exemple. Avec un décalage de $k = 3$, **CAPTUREZ IDEFIX!** devient **FDSWXUHC LGHILA!**

3. Programme une fonction `dechiffre_cesar_phrase(phrase, k)` pour le déchiffrement.

Indication. Deux solutions : soit tu recommences tout, mais au lieu d'ajouter k (modulo 26) tu soustrais k (modulo 26) ou bien tu chiffrés le message avec un décalage de $-k$ (au lieu de $+k$).

Vérifie que, après avoir chiffré un message, tu le retrouves par déchiffrement !

4. Tu te places dans la peau d'un espion qui a intercepté un message codé par un chiffre de César, mais qui ne connaît pas la clé k . Programme une fonction `attaque_cesar` (`phrase`) qui affiche les déchiffrements, en testant toutes les clés k possibles.

Tu as intercepté le message envoyé par le camp Babaorum à Jules César :

HSFGJSEAP F S HDMK VW HGLAGF

Que va maintenant faire César ?

Cours 3 (Chiffrement par substitution).

Le chiffre de César est trop facile à attaquer, même si on ne connaît pas le décalage. Pour compliquer la tâche d'un espion, on introduit le **chiffrement par substitution**. À chaque lettre de l'alphabet en clair (ici en rouge, en majuscule), on associe une lettre au hasard (ici en vert, en minuscule). Voici un exemple :

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
y	k	c	o	d	m	f	j	g	z	a	x	r	n	b	u	t	q	i	p	h	w	e	s	v	l

- **Chiffrement.** On remplace chaque caractère du message par sa lettre substituée. Le message **BON-JOUR** devient **kbnzbhq**. Exercice : chiffre la phrase **AU REVOIR**.
- **Déchiffrement.** On fait l'opération inverse : **kbnkbn** donne **BONBON**. Exercice : déchiffre **ywd cdiyq**.
- Bien évidemment pour les lettres d'arrivée on aurait pu choisir un autre mélange. L'expéditeur et le destinataire du message doivent au préalable se mettre d'accord sur la substitution choisie.

Activité 2 (Chiffrement par substitution).

Objectifs : programmer le codage et le décodage d'un chiffrement par substitution.

On se donne une substitution par un alphabet de départ suivi de son remplacement. On prendra l'exemple suivant, défini par deux constantes globales :

```
Alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Melange = "ykcodmfjgzaxrnbutqiphwesvl"
```

1. Programme une fonction `chiffre_substitution_caractere(car)` qui renvoie le caractère de l'alphabet chiffré par substitution.

Indications.

- Récupère le rang i dans `Alphabet` du caractère à chiffrer.
- Renvoie le caractère correspondant au rang i de `Melange`.

Exemple. **A** devient **y** et **B** devient **k**.

2. Programme une fonction `chiffre_substitution_phrase(phrase)` qui renvoie la phrase codée par substitution.

Exemple. Après substitution **PAS DE POTION POUR OBELIX!** devient **uyi od ubpgbn ubhq bkdngx!**

3. Programme une fonction `dechiffre_substitution_phrase(phrase)` pour le déchiffrement.

Cours 4 (Attaque statistique).

Pour un espion qui intercepterait un message codé, sans connaître la substitution choisie, il n'est plus possible de tester toutes les possibilités. En effet, il y a $26 \times 25 \times 24 \times \dots \times 2 \times 1$ choix possibles pour l'alphabet mélangé, ce qui fait environ 4×10^{26} clés !

Statistiques.

Pour un texte assez long, les lettres n'apparaissent pas toutes avec la même fréquence. En français, les lettres les plus rencontrées sont dans l'ordre :

E S A I N T R U L O D C P M V Q G F H B X J Y Z K W

La **fréquence d'apparition d'une lettre** est donnée par la formule :

$$\text{fréquence d'apparition d'une lettre} = \frac{\text{nombre d'occurrences de la lettre}}{\text{nombre total de lettres}} \times 100$$

Dans un texte en français les fréquences sont proches de :

E	S	A	I	N	T	R	U	L	O	D
14.69%	8.01%	7.54%	7.18%	6.89%	6.88%	6.49%	6.12%	5.63%	5.29%	3.66%

Attaque.

On a intercepté un message, mais on ne connaît pas la substitution. Comment utiliser les statistiques pour décrypter le message ? Voici une méthode d'attaque : dans le texte chiffré, on cherche la lettre qui apparaît le plus, et si le texte est assez long, cela devrait être le chiffrement du **E**, la lettre qui apparaît ensuite dans l'étude des fréquences devrait être le chiffrement du **S**, puis le chiffrement du **A**, **I**, **N**, **T**... On obtient ainsi un déchiffrement partiel du message, sous la forme d'un texte à trous et il faut ensuite deviner les lettres manquantes.

Un exemple.

Par exemple, déchiffrons la phrase :

jm dw ug jddbhbwm y jmlj cjtltjdj

On compte les apparitions des lettres :

j : 7 d : 5 m : 3 b, l, t, w : 2

On suppose donc que le **j** crypte la lettre **E**, le **d** la lettre **S**, ce qui donne :

E* S* ** ESS*** * E**E *E**E*SES**

Ensuite la lettre qui apparaît le plus est le **m**. D'après les fréquences, elle devrait correspondre à **A**, **I**, **N** ou **T**. Ainsi le premier mot serait **EA**, **EI**, **EN** ou **ET**. Seuls les deux derniers sont des mots valides.

Si **m** → **N**, la phrase se déchiffre en :

EN S* ** ESS***N * EN*E *E**E*SES**

ce qui n'est pas très clair, alors qu'avec **m** → **T** c'est mieux !

ET S* ** ESS***T * ET*E *E**E*SES**

En cherchant où placer les lettres les plus fréquentes suivantes (**A**, **I**, **N**) puis les autres, avec un peu de patience et de bon sens, on décrypte le message :

ET SI ON ESSAYAIT D ETRE HEUREUSES

Activité 3 (Attaque statistique).

Objectifs : utiliser la fréquence d'apparition des lettres pour décrypter un message en essayant de deviner la substitution.

1. Substitution. Programme une fonction :

`substitution(phrase,Alphabet_depart,Alphabet_arrivee)`

qui substitue dans la phrase donnée les lettres de l'alphabet de départ par celles de l'alphabet d'arrivée.

Indications.

- C'est presque la même fonction que l'activité précédente, sauf qu'ici on a plus de souplesse sur les alphabets qui sont passés en paramètres.
- En particulier `substitution(phrase,Alphabet,Melange)` fait la

même chose que `chiffre_substitution_phrase(phrase)` alors que `substitution(phrase,Melange,Alphabet)` fait la même chose que `dechiffre_substitution_phrase(phrase)`.

- En particulier la fonction `substitution()` doit permettre un déchiffrement partiel d'une phrase. Voir l'exemple ci-dessous.

Exemple (Utilisation pour un décryptage partiel.).

- On veut décrypter :

jdolwt jd tm vb ?

- Admettons que l'on ait identifié que **j** codait **E**, **d** codait **S**, **m** codait **T**.
- On définit donc un alphabet de départ et celui (partiel) d'arrivée :

`Alphabet_depart = "abcdefghijklmnopqrstuvwxyz"`

`Alphabet_arrivee = "...S....E..T....."`

- Alors avec `phrase = "jdolwt jd tm vb ?"`, la commande `substitution(phrase,Alphabet_depart,Alphabet_arrivee)` renvoie :

"ES...T ES T. .. ?"

- Il reste à deviner **ESPRIT ES TU LA?**

2. Statistiques.

Programme une fonction `statistiques(phrase)` qui calcule et renvoie le nombre d'apparitions des lettres dans une phrase.

Indications.

- On suppose que la phrase est écrite en minuscules, on ne tient pas compte des autres caractères.
- La fonction renvoie la liste des nombres correspondant à chaque lettre de l'alphabet. Par exemple [3, 0, 2, ...] signifie qu'il y a 3 lettres **a**, pas de lettres **b**, 2 lettres **c**, etc.

Déduis-en une fonction `affiche_statistiques(phrase)` qui affiche proprement les lettres qui apparaissent dans la phrase avec leur nombre d'apparitions.

3. **Fréquences.** Modifie tes fonctions précédentes pour écrire deux fonctions `frequences(phrase)` et `affiche_frequences(phrase)` qui calculent et affichent les fréquences d'apparitions des lettres (on ne tient compte que des lettres en minuscules).

4. **Énigmes.** Essaie de décrypter les trois citations suivantes. Chacune a été chiffrée par une substitution différente.

Les frères Goncourt :

ay dmymndmnlxlv vdm ay shvjnvvhv fvd dznvgzvd ngvcyzmvd

Charles Darwin :

**apy pywpfpy tdv ydjsvspng np ybng woy apy pywpfpy apy wady lbjgpy nv apy wady
vngpaavzpngpy movy fpaapy tdv y okowgpng ap mvpdc odc fionzpmpngy**

Albert Einstein :

**kw yjnzfcn, i nmy lqwpz zp mwcy yzqy ny lqn fcnp pn azpiyczppn. kw ofwyclqn, i nmy lqwpz
yzqy azpiyczppn ny lqn onfmzppn pn mwcy ozqflqzc. cic, pzqm wszpm fnqpc yjnzfcn ny
ofwyclqn : fcnp pn azpiyczppn ny onfmzppn pn mwcy ozqflqzc !**

Cours 5 (Chiffrement de Vigenère).

Un des principaux défauts du chiffre de César (et du chiffrement par substitution) est qu'une lettre (par exemple A) est toujours chiffrée par la même lettre (par exemple D). Le chiffrement de Vigenère est une version améliorée du chiffre de César.

On regroupe d'abord les lettres de notre message par blocs, par exemple ici par blocs de longueur 3 :

IL ETAIT UNE FOIS

devient

ILE TAI TUN EFO IS

(les espaces sont purement indicatifs, dans la première phrase ils séparent les mots, dans la seconde ils séparent les blocs).

Si n est la longueur d'un bloc, alors on choisit une clé constituée de n nombres de 0 à 25 : $[k_1, k_2, \dots, k_n]$.

Le chiffrement consiste à effectuer un chiffrement de César, dont le décalage dépend du rang de la lettre dans le bloc :

- un décalage de k_1 pour la première lettre de chaque bloc,
- un décalage de k_2 pour la deuxième lettre de chaque bloc,
- ...
- un décalage de k_n pour la n -ème et dernière lettre de chaque bloc.

Pour notre exemple, si on choisit comme clé $[4, 2, 3]$ alors pour le premier bloc « **ILE** » :

- un décalage de 4 pour **I** donne **M**,
- un décalage de 2 pour **L** donne **N**,
- un décalage de 3 pour **E** donne **H**,

Ainsi « **ILE** » devient « **MNH** ». On recommence avec le bloc « **TAI** » qui devient « **XCL** ». Le chiffrement complet donne :

MNH XCL XWQ IHR MU

autrement dit la phrase chiffrée est :

MN HXCLX WQI HRMU

Exercice. Chiffre la phrase **UNE PRINCESSE GEEK** avec le chiffrement de Vigenère et la même clé $[4, 2, 3]$.

Déchiffrement. Pour déchiffrer, il suffit d'inverser la clé. C'est-à-dire que c'est la même procédure que le chiffrement mais avec la clé $[-k_1, -k_2, \dots, -k_n]$. Par exemple l'opposé de la clé $[4, 2, 3]$ est la clé $[-4, -2, -3]$. Comme on travaille modulo 26, cette dernière clé vaut aussi $[22, 24, 23]$.

Exercice. Déchiffre la phrase **QKOSW HX VLRVLR**, chiffrée avec la même clé $[4, 2, 3]$.

Activité 4 (Chiffrement de Vigenère).

Objectifs : programmer le chiffrement de Vigenère et éventuellement trouver une attaque.

Programme une fonction `chiffre_vigenere(phrase, cle)` qui chiffre la phrase donnée selon le chiffrement de Vigenère, pour la clé donnée sous la forme d'une liste de décalages, $cle = [k_1, k_2, \dots, k_n]$.

Exemple. La phrase **AAA ABC** avec la clé $[1, 2, 3]$ renvoie la phrase **BCD BDF**.

Indications.

- Il faut décaler chaque lettre selon un décalage k_i comme pour le chiffre de César.
- Pour savoir quel indice i convient, tu peux utiliser un compteur i , initialisé à 0, auquel tu ajoutes 1 à chaque lettre de l'alphabet rencontrée, compteur qui prend ses valeurs modulo n (qui est la longueur de la clé).

Bonus. Attaque du chiffrement de Vigenère.

Essaie de décrypter le message suivant (et trouve son auteur) qui a été codé par un chiffrement de Vigenère avec une clé (inconnue !) de longueur 4.

DL ZHGIVUEL OD UL LQK TYDVL OIL XEU DLC YEIOSASOI K VXJ K BBI WA PYWYC T WBH
QDVBI IBO BWZ QUFZ S D L L V B A N O D L Z C E F A O C H S S I V L N E M A O I

Le compte est bon

Qui n'a jamais rêvé d'épater sa grand-mère en gagnant à tous les coups au jeu « Des chiffres et des lettres » ? Une partie du jeu est « Le compte est bon » dans lequel il faut atteindre un total à partir de chiffres donnés et des quatre opérations élémentaires. L'autre partie du jeu est « Le mot le plus long », cette fois il faut trouver le plus long mot français à partir d'un tirage de lettres. Pour ces deux jeux les ordinateurs sont plus rapides que les humains, il ne te reste plus qu'à écrire les programmes !

La résolution complète du « Compte est bon » utilise un algorithme récursif.

Quatre activités sont proposées :

1. on commence par générer un tirage de nombres ;
2. on continue par un algorithme très simple qui teste beaucoup de solutions mais ne fonctionne pas pour certains cas ;
3. on continue avec un problème plus simple : atteindre une somme fixée avec des nombres donnés ;
4. enfin on mixe nos deux activités pour résoudre complètement notre jeu.

Cours 1 (Le compte est bon).

On se donne une liste d'entiers et un total à atteindre. Il faut réaliser ce total à l'aide des opérations « + », « - », « × » et « / ». Exemple :

$$[1, 3, 6, 8, 75, 100] \quad T = 524$$

Une solution : $6 - 1 = 5$, puis $5 \times 100 = 500$, puis $3 \times 8 = 24$, $500 + 24 = 524$!

Voici les règles du jeu :

- le total à atteindre est un entier tiré au hasard entre 100 et 999,
- on dispose de 6 plaques tirées au hasard sur lesquelles sont inscrits des entiers,
- les plaques sont tirées parmi 28 plaques (elles sont en double) :

1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 25, 25, 50, 50, 75, 75, 100, 100

- chacune des 6 plaques ne peut être utilisée qu'une seule fois (mais on n'est pas obligé d'utiliser toutes les plaques),
- les seules opérations autorisées sont « + », « - », « × » et « / »,
- les calculs ne se font qu'avec des entiers positifs (les soustractions doivent donner un résultat positif, les divisions doivent avoir un reste nul).

Il n'est pas toujours possible de trouver le bon résultat. Dans ce cas, on cherche à s'approcher le plus possible du total.

Activité 1 (Tirage).

Objectifs : programmer le tirage du total et des plaques.

1. Programme une fonction `tirage_total()` qui renvoie un entier au hasard compris entre 100 et 999.

Indication. Utilise la fonction `randint(a,b)` du module `random`.

2. Programme une fonction `tirage_chiffres()` qui renvoie une liste de 6 entiers tirés au hasard parmi la liste des plaques possibles.

Exemple de tirage : [3, 5, 7, 7, 25, 100].

Indication. Il s'agit d'un tirage sans remise : une plaque tirée est ensuite écartée pour le reste du tirage. Comme l'ordre n'a pas d'importance pour le jeu, on peut renvoyer une liste ordonnée.

3. Programme une fonction `operation(a,b,op)` où a et b sont deux nombres et `op` est un caractère parmi '+', '-', '*', '/' et qui renvoie le calcul demandé parmi $a + b$, $a - b$, $a \times b$, a/b .

Exemple. `operation(7,5,'-')` renvoie $7 - 5 = 2$.

Activité 2 (Recherche basique).

Objectifs : programmer simplement une recherche brutale mais pas complète.

Solutions séquentielles.

Dans un premier temps on va chercher seulement les solutions séquentielles, dans lesquelles les calculs s'enchaînent de gauche à droite. Par exemple avec : [2, 3, 4, 5]

- on peut obtenir séquentiellement 29 par le calcul

$$((2 + 3) \times 5) + 4 = 29$$

c'est-à-dire que l'on fait les calculs un par un en repartant à chaque fois du résultat précédent : $2 + 3 = 5$, puis $5 \times 5 = 25$, puis $25 + 4 = 29$;

- par contre on ne peut obtenir séquentiellement le total de 49, même si on pourrait l'obtenir par les règles normales : d'une part $2 + 5 = 7$, d'autre part $3 + 4 = 7$ et on mixe ces deux sous-résultats $7 \times 7 = 49$.

Avec deux chiffres.

Voici comment calculer toutes les opérations $c_1 + c_2$ et $c_1 \times c_2$ à partir d'une liste de chiffres et afficher si le total souhaité (ici 18) est atteint.

```
chiffres = [2,3,5,6,8,10]          # Plaques disponibles
total = 18                         # Objectif

chiffres1 = list(chiffres)         # Copie
for c1 in chiffres1:              # Premier chiffre
    chiffres2 = list(chiffres1)   # Copie
    chiffres2.remove(c1)          # Retirer la plaque déjà choisie

    for op in ['+', '*']:          # Opérations

        for c2 in chiffres2:      # Second chiffre

            calcul = operation(c1,c2,op) # Résultat du calcul

            if calcul == total:     # Total atteint ?
                print("Trouvé !",c1,op,c2,'=',calcul)
```

1. Analyse ces ligne de code et transforme-le en une fonction `deux_plaques(total, chiffres)` qui gère cette fois les quatre types d'opérations. Est-ce que le total de 18 peut être atteint avec les chiffres disponibles [2, 3, 5, 6, 8, 10]? Si oui, de combien de façons différentes?
2. Modifie ton programme pour rechercher tous les calculs possibles avec trois plaques, c'est-à-dire $(c_1 \square c_2) \circ c_3$, où $\square$ et $\circ$ sont des opérations parmi les quatre opérations autorisées.
3. Persévère avec une fonction `recherche_basique(total, chiffres)` qui teste si on peut obtenir le total demandé à l'aide des 6 plaques données par la liste `chiffres`.

Indications. Il faut imbriquer beaucoup de boucles ! On ne s'occupe pas encore des règles strictes du « Compte est bon », on s'autorise des soustractions négatives et des divisions non entières.

Exemples :

- Laisse la machine obtenir 809 à partir de [2, 3, 6, 8, 75, 100].
 - Atteins 779 avec [5, 6, 7, 8, 25, 50].
 - Peux-tu trouver 773 avec [2, 4, 6, 8, 10, 50]? Et 769 avec ces mêmes chiffres?
4. On se donne un tirage de 6 plaques. Combien l'algorithme teste-t-il de combinaisons? Quel ordre de grandeur de durée met Python pour tester toutes ces possibilités? (Donne la réponse sous la forme 0.1 seconde, 1 seconde, 10 secondes, 100 secondes, 1000 secondes, 10 000 secondes...)

Pour le nombre de combinaisons, aide-toi des calculs plus simples suivants :

- Combien y a-t-il de combinaisons avec une seule opération $c_1 \square c_2$? Réponse : il y a 6 choix pour c_1 (une plaque parmi les 6), il y a 4 opérations possibles, et enfin il y a 5 choix pour c_2 (une plaque parmi les 5 plaques restantes). Bilan :

$$6 \times 4 \times 5 = 120 \text{ combinaisons.}$$

- Avec deux opérations $(c_1 \square c_2) \circ c_3$, il y a

$$6 \times 4 \times 5 \times 4 \times 4 = 1920 \text{ combinaisons.}$$

Activité 3 (Parcours d'arbre).

Objectifs : résoudre des problèmes en parcourant des arbres. Cette activité utilise la récursivité.

Atteindre une somme.

On se donne une liste de nombres, par exemple [5, 7, 11, 13]. À partir de ces nombres on doit obtenir une somme S fixée, par exemple $S = 28$. Ici c'est possible par exemple :

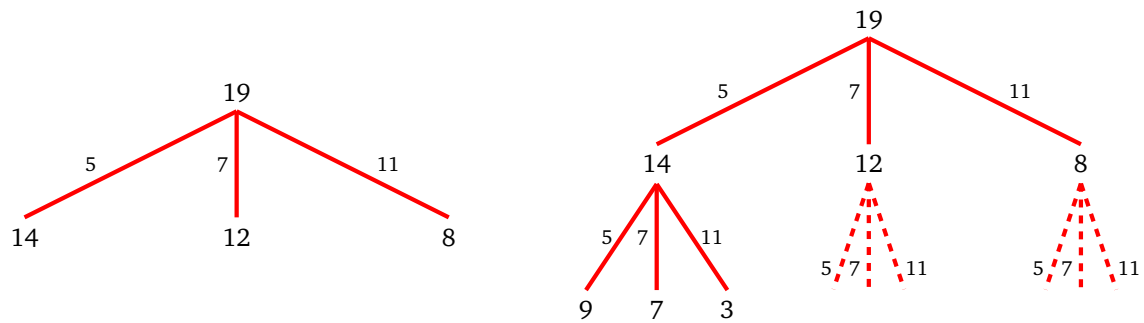
$$5 + 5 + 7 + 11 = 28$$

Remarques. Ici on peut utiliser plusieurs fois le même nombre ; on n'est pas obligé d'utiliser tous les nombres ; il n'y a pas toujours de solution ; il peut aussi y avoir plusieurs solutions.

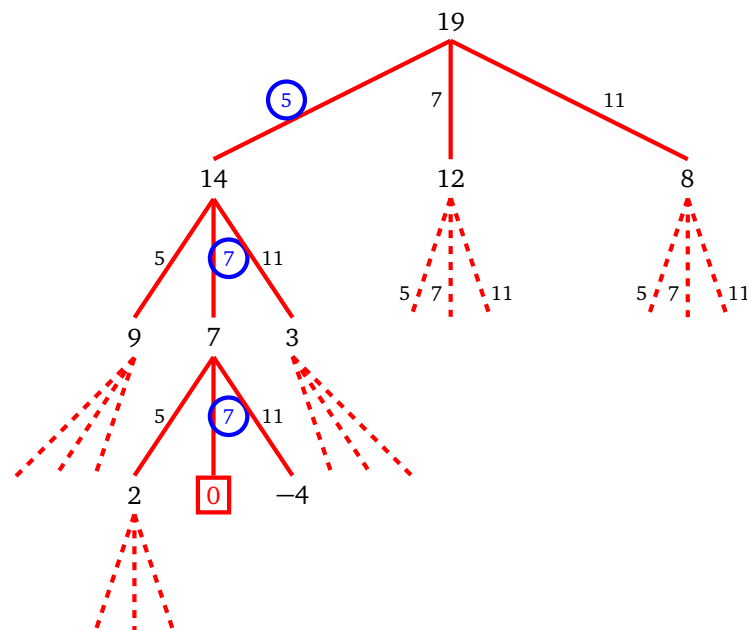
Pour résoudre ce problème, on teste chacun des entiers possibles, ici 5, puis 7, puis 11, puis 13. Commençons par tester 5. Si on choisit 5 alors la somme qu'il reste à atteindre est $S' = S - 5 = 23$. On a donc simplifié le problème. Si on avait choisit 7 alors la somme qu'il reste à atteindre serait $S' = S - 7 = 21$, etc.

Cela s'écrit sous la forme d'un arbre : les arêtes correspondent à un entier, les sommets à la somme qu'il reste à atteindre. L'arbre se termine quand la somme à atteindre est 0 (on a gagné) ou est négative (cette branche ne fonctionne pas).

Voyons l'exemple de la somme $S = 19$ à atteindre avec les entiers [5, 7, 11]. Sur l'arbre de gauche la somme S avec pour enfants les valeurs $S - 5$, $S - 7$ et $S - 11$. Sur l'arbre de droite on poursuit les calculs.



Si un sommet a une somme à atteindre négative c'est terminé, le choix testé ne convient pas. Si par contre la somme vaut 0, c'est que la somme est réalisée. Les entiers à choisir pour réaliser la somme initiale sont ceux lus sur les arêtes. Sur la figure ci-dessous, on arrive à obtenir un sommet dont la somme est 0 (dans le carré), les arêtes qui permettent d'atteindre ce sommet portent les poids 5, puis 7, puis 7 (dans les cercles). Et effectivement $5 + 7 + 7 = 19$.



Programme une fonction `atteindre_somme(S, chiffres)` qui renvoie une solution possible. Par exemple `atteindre_somme(53, [6, 7, 15])` renvoie `[6, 6, 6, 6, 7, 7, 15]` et effectivement $6 + 6 + 6 + 6 + 7 + 7 + 15 = 53$.

Voilà l'algorithme proposé qui correspond au parcours d'un arbre jusqu'à atteindre la somme S .

Algorithme.

- — Entête : `atteindre_somme(S, chiffres)`
 - Entrée : une somme S et une liste d'entiers strictement positifs.
 - Sortie : une combinaison, sous la forme d'une liste, permettant de réaliser la somme ou bien `None` en cas d'échec.
 - Action : fonction récursive.
- *Cas terminaux.* Si $S = 0$ alors renvoyer la liste vide `[]` qui va plus tard contenir le parcours gagnant. Si $S < 0$, renvoyer `None`.
- *Cas général.* Pour chaque élément x de la liste des chiffres :
 - par un appel récursif, on définit `parcours` comme le résultat de `atteindre_somme(S-x, chiffres)`,
 - si `parcours` ne vaut pas `None` alors on lui ajoute x en tête de liste, puis on renvoie `parcours`.
- Renvoyer `None` (c'est le cas uniquement si rien n'a été renvoyé au cours des instructions précédentes).

Activité 4 (Le compte est bon).

Objectifs : programmer la solution du « Compte est bon » en s'inspirant des deux activités précédentes.

Programme une fonction `compte_est_bon(total, chiffres)` qui renvoie une solution possible au problème d'atteindre le total donné avec une liste de nombres donnés. Par exemple avec les nombres `chiffres = [2, 3, 7, 9, 9, 25]` on peut atteindre `total = 457`. Avec ces données la fonction `compte_est_bon(total, chiffres)` renvoie :

`['2+3=5', '5+9=14', '7+25=32', '14*32=448', '9+448=457', '457']`

Ce qui donne le détail des opérations à effectuer :

$$(2 + 3 + 9) \times (7 + 25) + 9 = 457$$

Il fallait y penser !

Voici l'algorithme qui permet de résoudre le problème. Il est préférable d'avoir bien compris les deux activités précédentes.

Algorithme.

- — Entête : `compte_est_bon(total, chiffres)`
 - Entrée : un total à atteindre et une liste d'entiers strictement positifs.
 - Sortie : une liste d'instructions permettant d'atteindre le total ou bien `None` en cas d'échec.
 - Action : fonction récursive.
- *Cas terminaux.* Si le total à atteindre est un élément de la liste `chiffres` alors renvoyer la liste `[total]` qui contient ce total comme seul élément. Si la liste `chiffres` ne contient aucun élément ou bien un seul élément (qui n'est pas le total) alors renvoyer `None`.
- *Cas général.* Ordonner la liste des chiffres *du plus grand au plus petit*.
 Pour chaque chiffre c_1 de la liste, pour chaque chiffre c_2 plus loin dans la liste, pour chaque opération parmi « + », « - », « × » et « / » :
 - former une nouvelle liste `nouv_chiffres` à partir de `chiffres` en retirant c_1 et c_2 ,
 - noter `calcul` le résultat de l'opération $c_1 + c_2$ ou $c_1 \times c_2, \dots$ selon l'opération,
 - on ne continue que si `calcul` est strictement positif et si c'est bien un entier,
 - on ajoute `calcul` à la liste `nouv_chiffres`,
 - par un appel récursif, on définit `parcours` comme le résultat de `compte_est_bon(total, nouv_chiffres)`,
 - si `parcours` ne vaut pas `None` alors c'est une liste d'instructions et on lui ajoute en tête de liste une instruction sous la forme d'une chaîne de caractères du type « $c_1 + c_2 = \text{resultat}$ » ou « $c_1 \times c_2 = \text{resultat}$ », ..., puis on renvoie `parcours`.
- Renvoyer `None` (c'est le cas uniquement si rien n'a été renvoyé au cours des instructions précédentes).

Exemple. Il faut bien comprendre qu'à chaque appel de la fonction, le total à atteindre reste le même, mais que la liste des chiffres change et diminue. Voyons un exemple dans lequel il faut atteindre 27 avec les chiffres [4, 5, 6, 7]. L'appel de la fonction est `compte_est_bon(27, [4, 5, 6, 7])`. Dans le corps de cette fonction, à un moment, on va tester l'opération 4×5 , on retire alors 4 et 5 de la liste des chiffres (car on n'a plus le droit de les utiliser), mais en échange on rajoute le résultat 20 ($= 4 \times 5$) que l'on peut utiliser dans la suite. Il s'agit donc maintenant d'atteindre 27 mais avec les chiffres [20, 6, 7]. L'appel récursif est donc `compte_est_bon(27, [20, 6, 7])`. Lors de ce nouvel appel, on va faire l'opération $20 + 7$ (nous savons déjà que cela va être bon), on remplace les chiffres 20 et 7 par le résultat 27, et on effectue l'appel `compte_est_bon(27, [27, 6])`. Comme l'un des chiffres est le total à atteindre, c'est un cas terminal gagnant ! La fonction renvoie l'historique des calculs « $4 \times 5 = 20$ », « $20 + 7 = 27$ ».

Indications.

- Ordonner la liste des chiffres du plus grand au plus petit est important. Cela permet de faire les opérations dans le bon sens : $7 - 5$ et pas $5 - 7$, $6/3$ et pas $3/6$.
- Voici comment tester si un nombre est un entier : `isinstance(x, int)`.

Voici des exemples à calculer :

- avec les chiffres [2, 4, 5, 6, 8, 10] il est possible d'atteindre 850 et 852 mais pas 851,
- dresse la liste des totaux compris entre 100 et 999 impossibles à atteindre avec ces chiffres,
- pour les chiffres [1, 2, 5, 7, 75, 100] vérifie que l'on peut atteindre tous les totaux possibles entre 100 et 999 (attention les calculs deviennent longs !),
- l'un des pire tirages est [10, 10, 25, 50, 75, 100] pour lequel il y a plus de totaux irréalisables que de totaux possibles.

Le mot le plus long

La seconde partie du jeu « Des chiffres et des lettres » est le « Le mot le plus long ». Il s'agit simplement de trouver le mot le plus grand à partir d'un tirage de lettres. Pour savoir si un mot est valide, on va utiliser une longue liste des mots français.

Activité 1 (Chercher un mot).

Objectifs : vérifier si un mot donné est bien un mot de la langue française. On va extraire ces mots d'un fichier afin de créer un répertoire qui contient tous les mots admissibles.

Préalable. Il faut récupérer un des deux fichiers suivants qui contient des mots français :

- `repertoire_francais_simple.txt` qui contient 20 239 mots les plus courants (de **ABAISSER** à **ZYGOTE**),
- ou `repertoire_francais_tout.txt` qui contient une liste complète de 131 896 mots français de **A** à **ZYGOMATIQUES**.

Ces fichiers sont disponibles ici :

github.com/exo7math/python2-exo7

Les mots de ce fichier sont classés par ordre alphabétique. Ils sont en majuscules, sans accents : les seuls caractères autorisés sont les lettres **A** à **Z**. Des rappels pour lire/écrire un fichier texte sont donnés juste après cette activité.

1. **Lecture du répertoire.** Programme une fonction `lire_repertoire(fichier)` qui lit un fichier texte (pour nous un des deux fichiers `repertoire.txt`) et renvoie la (longue) liste de tous ces mots. On appelle « répertoire » la liste de tous ces mots français. Vérifie que tu obtiens le nombre de mots annoncé.

Dans la suite on veut décider si un mot donné est un mot français ou pas. On dit qu'un mot est valide s'il apparaît dans notre répertoire. Par exemple **COUCOU** est français car il apparaît dans le répertoire (c'est le mot numéro 4822 dans la liste simple et le numéro 27374 dans la liste complète, en commençant la numérotation à 0). Par contre **BOLOSS** n'y est pas et n'est donc pas considéré comme un mot français.

Il s'agit donc de décider si un mot donné est présent ou pas dans une liste. Python saurait faire cela très bien, à l'aide de l'opérateur `in` ou de la méthode `index()`. Ici on va programmer deux fonctions.

2. **Recherche séquentielle.**

Programme une fonction `recherche_basique(mot, liste)` qui renvoie le rang du mot dans la liste s'il est présent et `None` sinon. Vérifie que ta fonction est correcte en comparant avec la commande `liste.index(mot)`.

Méthode imposée. Utilise une boucle « tant que » qui parcourt un par un les éléments de la liste et les compare au mot cherché. Tu as seulement le droit d'effectuer des tests d'égalité entre deux chaînes de caractères.

Questions. Le mot **SERPENT** est-il dans notre répertoire, si oui à quelle place ? Et le mot **PYTHON** ?

Analyse. Si on cherche le rang d'un mot français alors, en moyenne, il faut parcourir la moitié de la liste avant de le trouver. Il faut donc effectuer en moyenne 10 000 tests dans le cas de la liste simple, ou environ 60 000 dans le cas de la liste complète. Si un mot n'appartient pas à la liste, on le sait seulement après avoir testé tous les éléments.

3. Recherche dichotomique.

On va profiter du fait que la liste soit ordonnée pour faire la recherche beaucoup plus rapidement. Programme une fonction `recherche_dichotomie(mot, liste)` qui renvoie de nouveau le rang du mot dans la liste s'il est présent et `None` sinon.

Algorithme de la dichotomie. Pour comprendre l'idée, voir le principe de la dichotomie dans le chapitre « Dérivée – Zéros de fonctions ».

Algorithme.

- — Entrée : un mot à trouver et une liste de mots.
- — Sortie : le rang du mot trouvé dans la liste ou `None` en cas d'échec. (Le rang d'une liste commence à 0.)
- $a \leftarrow 0$
- $b \leftarrow n - 1$ où n est la longueur de la liste.
- Tant que $b \geq a$, faire :
 - $k \leftarrow (a + b) // 2$
 - Si `mot` égal `liste[k]` alors renvoyer k .
 - Si `mot` vient après `liste[k]` dans l'ordre alphabétique alors faire $a \leftarrow k + 1$,
 - sinon faire $b \leftarrow k - 1$.
- Renvoyer `None` (c'est le cas uniquement si aucun mot n'a été trouvé dans la boucle précédente).

Ordre alphabétique. Python connaît l'ordre alphabétique ! Par exemple « 'INDIC' < 'INFO' » vaut « Vrai ». On peut comparer deux chaînes de caractères :

- `mot1 == mot2` vaut « Vrai » si les chaînes sont égales,
- `mot1 < mot2` vaut « Vrai » si `mot1` est avant `mot2` dans l'ordre alphabétique,
- `mot1 > mot2` vaut « Vrai » si `mot1` est après `mot2` dans l'ordre alphabétique.

Question. Est-ce que tu obtiens les mêmes résultats qu'auparavant pour les mots **SERPENT** et **PYTHON** ? (La réponse doit être « oui » !)

Analyse. Pour chercher un mot, on divise la liste en deux à chaque étape. Donc en k étapes on trouve un mot dans une liste de longueur 2^k . Ainsi si on a une liste de longueur n alors il faut environ $\log_2(n)$ étapes. Le nombre d'étapes devant être un entier c'est en fait $k = E(\log_2(n)) + 1$ (où $E(x)$ désigne la partie entière, qui est la commande `floor(x)` de Python). Si on compare avec la recherche séquentielle, l'amélioration est frappante :

Taille de la liste	Nb. max. d'étapes - Séquentielle	Nb. max. d'étapes - Dichotomie
n	n	$E(\log_2(n)) + 1$
Rép. simple $n = 20\,239$	20 239	15
Rép. complet $n = 131\,896$	131 896	18

Cours 1 (Les fichiers).

Voici un très bref rappel des instructions Python pour lire et écrire un fichier texte.

Lire un fichier. Ici on ouvre un fichier en lecture et on affiche à l'écran chaque ligne.

```
fic = open("mon_fichier.txt", "r")

for ligne in fic:
    print(ligne.strip())

fic.close()
```

La commande `ligne.strip()` renvoie la chaîne de caractères de la ligne sans les espaces de début et de fin, ni le saut de ligne.

Écrire un fichier. Voici comment écrire un fichier. Ici chaque ligne écrite est de la forme : Ligne numéro x.

```
fic = open("mon_fichier.txt", "w")

for i in range(100):
    ligne = "Ligne numéro " + str(i) + "\n"
    fic.write(ligne)
fic.close()
```

Cours 2 (Dictionnaire).

Un dictionnaire est un peu comme une liste, mais les éléments ne sont pas indexés par des entiers mais par une « clé ». Un **dictionnaire** est donc un ensemble de couples clé/valeur : à une **clé** est associée une **valeur**.

Exemple : un dictionnaire identifiant/mot de passe.

- Voici l'exemple d'un dictionnaire dico qui stocke des identifiants et des mots de passe :

```
dico = {'jean': 'rev1789', 'adele': 'azerty', 'jasmine': 'c3por2d2'}
```
- Par exemple 'adele' a pour mot de passe 'azerty'. On obtient le mot de passe comme on accéderait à un élément d'une liste par l'instruction :

```
dico['adele'] qui vaut 'azerty'.
```

- Pour ajouter une entrée on écrit :

```
dico['lola'] = 'abcdef'
```

- Pour modifier une entrée :

```
dico['adele'] = 'vwxyz'
```

- Maintenant la commande `print(dico)` affiche :

```
{'jean': 'rev1789', 'adele': 'vwxyz', 'jasmine': 'c3por2d2', 'lola': 'abcdef'}
```

- Le parcours d'un dictionnaire se fait par une boucle « pour ». Par exemple, la boucle suivante affiche l'identifiant et le login de tous les éléments du dictionnaire :

```
for prenom in dico:
    print(prenom + " a pour mot de passe " + dico[prenom])
```

- Attention : il n'y a pas d'ordre dans un dictionnaire. Tu ne contrôles pas dans quel ordre les éléments sont traités.

Commandes principales.

- Définir un dictionnaire `dico = {cle1:valeur1, cle2:valeur2,...}`

- Récupérer une valeur : `dico[cle]`
- Ajouter une valeur : `dico[new_cle] = valeur`
- Modifier une valeur : `dico[cle] = new_valeur`
- Taille du dictionnaire : `len(dico)`
- Parcourir un dictionnaire : `for cle in dico:` et dans la boucle on accède aux valeurs par `dico[cle]`
- Tester si une clé existe : `if cle in dico:`
- Dictionnaire vide : `dico = {}`, utile pour initialiser un dictionnaire dans le but de le remplir ensuite.

Des commandes un peu moins utiles :

- Liste des clés : `dico.keys()`
- Liste des valeurs : `dico.values()`
- On peut récupérer les clés et les valeurs pour les utiliser dans une boucle :

```
for cle,valeur in dico.items():
    print("Clé :", cle, " Valeur :", valeur)
```

Activité 2 (Dictionnaire avec Python).

Objectifs : s'initier à la manipulation des dictionnaires en Python.

1. On reprend le dictionnaire des identifiants/mots de passe :

```
dico = {'jean':'rev1789', 'adele':'vwxyz', 'jasmine':'c3por2d2',
        'lola':'abcdef'}
```

- (a) Ajoute 'angela' avec le mot de passe 'qwerty'. Affiche alors le dictionnaire et sa longueur.
- (b) Programme une fonction `affiche_mot_de_passe(prenom)` qui affiche soit « untel a pour mot de passe ... » ou bien « untel n'a pas de mot passe ».

2. Voici les prénoms/âges des enfants d'une classe.

Clé (prénom)	Valeur (âge)
'zack'	8
'paul1'	5
'eva'	7
'paul2'	6
'zoe'	7

Note que l'on peut avoir deux valeurs identiques, mais pas deux clés identiques (d'où 'paul1' et 'paul2').

- (a) Définis un nouveau dictionnaire `dico` qui correspond à ces données. Les valeurs sont ici des entiers.
- (b) Programme une boucle qui calcule la somme des âges, puis calcule la moyenne des âges.

3. Voici les notes d'un élève par matière :

Clé (matière)	Valeur (liste de notes)
'maths'	[13, 15]
'anglais'	[16, 12, 14]
'sport'	[17]

- (a) Pars d'un dictionnaire `notes = {}` vide, puis complète matière par matière le dictionnaire. Les valeurs sont ici des listes d'entiers.
- (b) Introduis une nouvelle matière 'python' avec les notes 18 et 17.
- (c) Ajoute la note de 16 en 'maths'. (Il s'agit juste d'ajouter un élément à la liste `dico['maths']`.)

Activité 3 (Anagrammes).

Objectifs : trouver tous les anagrammes de la langue française.

- Deux mots sont des **anagrammes** s'ils ont les mêmes lettres, mais dans des ordres différents. Par exemple **CRIME** et **MERCI** ou bien **PRIERES** et **RESPIRE**.
- L'**indice** d'un mot est la suite ordonnée de ses lettres. Par exemple l'indice du mot **KAYAK** est **AAKKY**.
- Deux mots forment des anagramme exactement lorsqu'ils ont des indices identiques. Par exemple **CRIME** et **MERCI** ont bien le même indice **CEIMR**.

1. Programme une fonction `calculer_indice(mot)` qui renvoie l'indice du mot. Par exemple `calculer_indice("KAYAK")` renvoie "AAKKY".

Indications minimalistes. Tu peux utiliser `join()`, `list()`, `sort()` dans le désordre.

2. Programme une fonction `sont_anagrammes(mot1,mot2)` qui teste si les deux mots donnés sont des anagrammes. Avec **CHIEN** et **NICHE** la fonction renvoie « Vrai ».
3. Programme une fonction `dictionnaire_indices_mots(liste)` qui à partir d'une liste de mots construit un dictionnaire ; dans ce dictionnaire les clés sont les indices, et à chaque indice est associé les mots de la liste correspondant.

Exemple. Voici une liste de mots :

```
['CRIME', 'COUCOU', 'PRIERES', 'MERCI', 'RESPIRE', 'REPRISE']
```

et voici le dictionnaire renvoyé par la fonction :

```
{
  'CEIMR': ['CRIME', 'MERCI'],
  'CCOOUU': ['COUCOU'],
  'EEIPRRS': ['PRIERES', 'RESPIRE', 'REPRISE']
}
```

Ainsi dans notre dictionnaire les mots sont regroupés par anagrammes et indexés par l'indice. Note que chaque valeur associée à un indice n'est pas un mot mais une liste de mots.

Méthode.

- Partir d'un dictionnaire vide `dico = {}`.
 - Pour chaque mot de la liste :
 - calculer l'indice du mot,
 - si cet indice n'existe pas déjà dans le dictionnaire, alors ajouter une nouvelle entrée : `dico[indice] = [mot]`,
 - si l'indice existe déjà, alors ajouter le mot à la liste existante : `dico[indice].append(mot)`.
4. À partir du répertoire (simple ou complet) des mots de la langue française, dresse la liste de tous les anagrammes possibles. Combien trouves-tu de classes d'anagrammes en tout ? Quel est l'anagramme supplémentaire à **PRIERES**, **RESPIRE**, **REPRISE** ? Trouve les anagrammes de **CESAR**.
 5. *Bonus.* Programme une fonction `fichier_indice_mots(fichier_in,fichier_out)` qui à partir du fichier de tous les mots français (`fichier_in`), écrit dans un fichier (`fichier_out`) les indices et les mots correspondants. Voici un extrait du fichier obtenu :

CEHO : ECHO
 CEHOP : CHOPE POCHE
 CEHOPR : CHOPER PROCHE
 CEHOPRS : PROCHES
 CEHOPS : POCHEs

Cours 3 (Le mot le plus long : règle du jeu).

Le jeu « Le mot le plus long » est très simple. On tire un certain nombre de lettres au hasard (de 7 à 10 lettres). Il s'agit de trouver un mot français ayant le plus de lettres possibles.

Par exemple avec les lettres [G, E, A, T, G, A, N], on peut former des mots de 5 lettres **AGENT**, **GAGNE**, **ETANG**, des mots de 6 lettres comme **GAGENT**, et des mots de 7 lettres comme **TANGAGE**.

Les lettres sont choisies au hasard, à l'aide d'un tirage sans remise. Certaines lettres (en particulier les voyelles) sont plus présentes que d'autres. Voici une constitution possible avec 59 plaques :

- Voyelles **A,E,I,O,U** : 5 plaques chacune,
- Consonnes fréquentes **B,C,D,F,G,H,L,M,N,P,R,S,T** : 2 plaques chacune,
- Consonnes rares **K,J,Q,V,W,X,Y,Z** : 1 plaque chacune.

Cours 4 (Le mot le plus long : stratégie).

Une mauvaise stratégie. Commençons par l'idée qui vient en premier pour trouver le mot le plus long. Imaginons que l'on a un tirage de 10 lettres. Notre idée est la suivante, on génère toutes les combinaisons de mots possibles à partir de ces 10 lettres, puis on teste un par un s'ils sont présents dans le répertoire des mots français. Quel est le problème ? Il y a $10! = 10 \times 9 \times 8 \times \dots \times 1 = 3\,628\,800$ combinaisons possibles (10 choix pour la première lettre, 9 choix pour la seconde...). En ensuite il faut tester si chaque mot est dans le répertoire, ce qui demande au moins 10 tests d'égalité (voir la première activité). Donc en tout plus de 30 millions de tests d'égalité ce qui est beaucoup, même pour Python !

Une bonne stratégie. Il vaut mieux partir des mots connus que des mots possibles, car il y en a moins. Voici le principe :

- Génération du dictionnaire (à faire une fois pour toute, voir l'activité 3) :
 - on part de la liste des mots existants en français,
 - pour chaque mot on calcule son indice,
 - on obtient un dictionnaire indexé par les indices et qui contient tous les mots français possibles associés.
- Trouver le mot le plus long à partir d'une suite de lettres :
 - on commence par ordonner ces lettres,
 - puis voir si cela correspond à un indice de notre dictionnaire,
 - si c'est non il n'y a pas de mot,
 - si c'est oui la valeur associée à l'indice donne le ou les mots français.

Exemple. Avec le tirage de lettres **N, E, C, O, I**, on ordonne les lettres pour obtenir le candidat indice **CEINO**. On cherche dans le dictionnaire des indices/mots si cet indice existe, et ici on trouve qu'il existe et qu'il est associé au mot **ICONE**.

Analyse. La génération du dictionnaire n'est faite qu'une seule fois et nécessite 20 000 (ou 130 000) étapes. À chaque tirage, chercher est très simple, il s'agit juste de chercher si un indice existe dans la liste des 20 000 (ou 130 000) indices. Si la liste est ordonnée, on a vu que cela se fait en moins de 20 étapes. C'est donc quasi-immédiat.

Une difficulté. Prenons le tirage **X, E, C, S, I**, le candidat indice est **CEISX**, mais il n'est associé à aucun mot français. Il faut donc chercher des mots moins longs, par exemple sans le **X**, les lettres restantes forment l'indice **CEIS** qui est l'indice du mot **SCIE**. Conclusion : pour un tirage donné il faut aussi considérer tous les sous-tirages possibles.

Activité 4 (Le mot le plus long).

Objectifs : gagner à tous les coups au jeu « Le mot le plus long »

1. Programme une fonction `tirage_lettres(n)` qui renvoie une liste de n lettres tirées au hasard.
2. Programme une fonction `liste_binaire(n)` qui renvoie toutes les listes possibles formées de n zéros et uns. Par exemple avec $n = 4$, la fonction renvoie la liste :

[[1,1,1,1], [1,1,1,0], [1,1,0,1], ..., [0,0,1,0], [0,0,0,1], [0,0,0,0]]

Indications. Tu peux utiliser l'écriture binaire des entiers. Pour k variant de 0 à $2^n - 1$:

- calculer l'écriture binaire de k (exemple `bin(7)` renvoie `'0b111'`),
- supprimer le préfixe `'0b'` et transformer le reste en une liste d'entier (ex. on obtient `[1,1,1]`),
- rajouter éventuellement des zéros pour obtenir la longueur souhaitée (ex. avec $n = 5$, on obtient `[0,0,1,1,1]`).

3. Reprends et modifie la fonction précédente pour obtenir une fonction `indices_depuis_tirage(tirage)` qui à partir d'une liste de lettres forme tous les indices possibles.

Par exemple avec le tirage `['A', 'B', 'C', 'L']` de $n = 4$ lettres, la fonction renvoie les $15 = 2^n - 1$ indices possibles (qui correspondent à tous les sous-tirages possibles) :

`['ABCL', 'ABC', 'ABL', 'ACL', 'BCL', 'AB', 'AC', 'BC',
'AL', 'BL', 'CL', 'A', 'B', 'C', 'L']`

Indication. Une liste de 0 et de 1 indique quelles lettres du tirage on conserve : on garde les lettres correspondant aux 1. Avec les lettres de départ `['A', 'B', 'C', 'L']` :

- `[1,1,1,1]` donne l'indice `'ABCL'` (on garde toutes les lettres),
- `[1,1,1,0]` donne l'indice `'ABC'`,
- `[1,1,0,1]` donne l'indice `'ABL'`,
- ...
- `[0,0,0,1]` donne l'indice `'L'`.

Il est important d'avoir ordonné les lettres du tirage par ordre alphabétique pour obtenir des indices valides.

4. Programme une fonction `mot_le_plus_long(tirage, dico)` qui permet de gagner au jeu. `tirage` est une liste de lettres et `dico` est un dictionnaire qui associe à des indices les mots français correspondant (c'est le dictionnaire de l'activité 3).

Indications.

- Calcule la liste des indices possibles à partir du tirage (c'est la question précédente).
- Ne conserve que les indices valides. Ce sont ceux qui apparaissent dans le dictionnaire (utilise le test `if indice in dico`).
- À partir des indices valides, trouve les mots français réalisables (ils sont directement donnés par `dico[indice]`).

Exemple. Avec notre tirage `['A', 'B', 'C', 'L']`, on trouve par exemple :

- indice valide `'ABC'`, mot associé `'BAC'`,

- indice valide 'ABL', mot associé 'BAL',
- indice valide 'ACL', mots associés 'CAL' et 'LAC'.

Comme aucun mot n'est associé à l'indice 'ABCL', il n'y a pas de mots français de longueur 4 avec ce tirage, les meilleurs mots possibles sont donc de longueur 3.

Voici des exemples à traiter :

- Trouve des mots avec le tirage de 7 lettres ['Z', 'M', 'O', 'N', 'U', 'E', 'G'].
- Trouve des mots avec le tirage de 8 lettres ['H', 'O', 'I', 'P', 'E', 'U', 'C', 'R'].
- Trouve des mots avec les 9 lettres ['H', 'A', 'S', 'T', 'I', 'D', 'O', 'I', 'T'].
- Trouve des mots avec les 10 lettres ['E', 'T', 'N', 'V', 'E', 'U', 'Z', 'O', 'V', 'N'].

Images et matrices

Le traitement des images est très utile, par exemple pour les agrandir ou bien les tourner. Nous allons aussi voir comment rendre une image plus floue, mais aussi plus nette ! Tout cela à l'aide des matrices.

Cours 1 (Convolution de matrices).

La convolution est l'action d'une matrice C de taille 3×3 sur une matrice M de taille quelconque et qui transforme la matrice M en une matrice N de même taille.

- **Principe.** Pour chaque élément m_{ij} de la matrice M de départ, on regarde la sous-matrice 3×3 qui entoure cet élément (en gris foncé ci-dessous), on va multiplier ces éléments par les éléments de la matrice C , tout ceci va donner un seul élément : le coefficient n_{ij} de la matrice N .

$$M = \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & m_{ij} & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{pmatrix} \quad C = \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix} \quad \longrightarrow \quad N = \begin{pmatrix} & & & & & \\ & & & & & \\ & & n_{ij} & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix}$$

- **Élément de convolution.** Voici comment calculer un élément de convolution. On multiplie chaque élément de la sous-matrice 3×3 de M par l'élément correspondant de C . La somme de ses produits donne l'élément de convolution :

$$n_{ij} = c_{00}a_{00} + c_{01}a_{01} + \cdots + c_{22}a_{22}$$

$$\begin{array}{l} \text{sous-matrice de } M \\ \begin{array}{ccc} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{array} \end{array} \quad C = \begin{pmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{pmatrix} \quad \longrightarrow \quad n_{ij} = c_{00} \cdot a_{00} + c_{01} \cdot a_{01} + \cdots + c_{22} \cdot a_{22}$$

- **Matrice obtenue.** On calcule donc la matrice N coefficient par coefficient, à chaque coefficient (symbolisés par un cercle) correspond donc une sous matrice 3×3 de la matrice M (figure ci-dessous à gauche). Comment faire pour les coefficients qui sont au bord de la matrice ? Si la sous-matrice déborde à droite elle repart à gauche (figure du centre), si elle déborde en bas, elle repart en haut, etc.

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{pmatrix} \quad \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{pmatrix} \quad \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{pmatrix}$$

Si on ne tenait pas compte des contraintes de bords, les coefficients de la sous-matrice autour du coefficient m_{ij} seraient donnés par la sous-matrice ci-dessous à gauche, mais pour tenir compte des débordements on raisonne à l'aide des modulus (sous-matrice de droite). Les indices i des lignes sont calculés modulo n (où n est le nombre de lignes de la matrice M), les indices j des colonnes sont calculés modulo p (où p est le nombre de colonnes de la matrice M) :

$m_{i-1,j-1}$	$m_{i-1,j}$	$m_{i-1,j+1}$	$m_{(i-1)\%n,(j-1)\%p}$	$m_{(i-1)\%n,j}$	$m_{(i-1)\%n,(j+1)\%p}$
$m_{i,j-1}$	$m_{i,j}$	$m_{i,j+1}$	$m_{i,(j-1)\%p}$	$m_{i,j}$	$m_{i,(j+1)\%p}$
$m_{i+1,j-1}$	$m_{i+1,j}$	$m_{i+1,j+1}$	$m_{(i+1)\%n,(j-1)\%p}$	$m_{(i+1)\%n,j}$	$m_{(i+1)\%n,(j+1)\%p}$

- Exemple.

$$M = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \quad C = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad N = \begin{pmatrix} 22 & 24 & 30 & 32 \\ 34 & 36 & 42 & 44 \\ 46 & 48 & 54 & 56 \end{pmatrix}$$

Par exemple voici le calcul du coefficient n_{11} :

$$\begin{aligned} n_{11} &= c_{00}m_{00} + c_{01}m_{01} + \dots + c_{22}m_{22} \\ &= 0 \times 1 + 1 \times 2 + 0 \times 3 + 1 \times 5 + 2 \times 6 + 1 \times 7 + 0 \times 9 + 1 \times 10 + 0 \times 11 \\ &= 36 \end{aligned}$$

Cours 2 (Modélisation de matrices).

- Une matrice est codée par un tableau, c'est-à-dire une liste de listes. Par exemple :

$$M = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]$$

représente la matrice :

$$M = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

- On accède à l'élément (i, j) par :

$$M[i][j]$$

- Une matrice M de n lignes et p colonnes est donc une liste de longueur n , chaque élément étant une liste de longueur p :

$$n = \text{len}(M) \quad p = \text{len}(M[0])$$

- Comme d'habitude la numérotation commence à 0, donc $0 \leq i < n$ et $0 \leq j < p$.
- On initialise une matrice avec des coefficients nuls ainsi :

$$M = [[0 \text{ for } j \text{ in range}(p)] \text{ for } i \text{ in range}(n)]$$

On peut ensuite remplir la matrice coefficient par coefficient par des commandes du type :

$$M[i][j] = \dots$$

Activité 1 (Convolution de matrices).

Objectifs : calculer une convolution.

1. Programme une fonction `afficher_matrice(M)` qui réalise un affichage propre d'une matrice sur le terminal de l'écran.

Voici un exemple d'affichage :

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Indications.

- `print('{:3d}'.format(x), end=" ")` affiche un entier sur 3 caractères.
 - `print('{0:.3f}'.format(x), end=" ")` affiche un nombre flottant avec 3 décimales après la virgule.
2. Programme une fonction `element_convolution(C,M)` qui à partir de deux matrices C et M de taille 3×3 calcule l'élément de convolution :

$$\gamma = c_{00}m_{00} + c_{01}m_{01} + \dots + c_{22}m_{22}$$

Autrement dit γ est la somme des produits des coefficients entre C et M . Par exemple pour :

$$C = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 5 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

L'élément de convolution est

$$\gamma = 1 \times 1 + 1 \times 2 + 1 \times 3 + 1 \times 4 + 5 \times 5 + 1 \times 6 + 1 \times 7 + 1 \times 8 + 1 \times 9 = 65.$$

3. Programme une fonction `convolution(C,M)` qui calcule la matrice N de convolution de C sur M .
 - C est une matrice 3×3 .
 - M est une matrice de taille quelconque $n \times p$.
 - $N = \text{convolution}(C, M)$ est aussi de taille $n \times p$ et chaque coefficient n_{ij} est l'élément de convolution de C avec M'_{ij} , la sous-matrice 3×3 de M autour du coefficient en (i, j) .
 - N'oublie pas que pour la sous-matrice il y a des effets de bords (voir les formules avec les modulus dans le cours ci-dessus).

Par exemple

$$C = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad M = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \quad \text{donnent} \quad N = \begin{pmatrix} 9 & 10 & 11 & 12 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

Avec cette matrice C , la convolution sur M correspond à faire descendre chaque coefficient d'une ligne (et comme on travaille avec les modulus, un coefficient de la dernière ligne se retrouve sur la première ligne).

4. Utilise ou modifie la fonction précédente en une fonction `convolution_entiere(C,M)` qui renvoie une matrice dont les coefficients sont arrondis à des entiers, limités à 255, les coefficients négatifs étant ramenés à zéros.

Exemple.

$$C = \begin{pmatrix} 0 & -\frac{1}{2} & 0 \\ -\frac{1}{2} & 3 & -\frac{1}{2} \\ 0 & -\frac{1}{2} & 0 \end{pmatrix} \quad M = \begin{pmatrix} 101 & 102 & 103 & 104 \\ 201 & 151 & 101 & 51 \\ 50 & 100 & 150 & 200 \end{pmatrix}$$

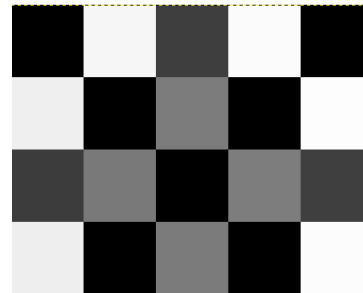
donnent la convolution N et la convolution entière N' :

$$N = \begin{pmatrix} 74.5 & 78.5 & 80.5 & 84.5 \\ 426.5 & 201.0 & 75.5 & -150.0 \\ -151.0 & 73.5 & 198.0 & 422.5 \end{pmatrix} \quad \text{et} \quad N' = \begin{pmatrix} 74 & 78 & 80 & 84 \\ 255 & 201 & 75 & 0 \\ 0 & 73 & 198 & 255 \end{pmatrix}$$

Cours 3 (Format d'image « pgm »).

Le format d'image « pgm » est un format très simple de fichier texte qui permet de décrire une image. Voici un exemple très simple : à gauche le fichier et à droite l'image correspondante.

```
P2
5 4
255
 0 255 64 255 0
255 0 127 0 255
64 127 0 127 64
255 0 127 0 255
```

**Explications.**

- P2 est le code pour le format d'image en niveau de gris (P1 désigne les images en noir et blanc, P3 les images en couleurs).
- Les entiers 5 et 4 sont le nombre de colonnes suivi du nombre de lignes de l'image (ici 5 colonnes et 4 lignes).
- 255 désigne le découpage en niveau de gris : ici de 0 (noir) à 255 (blanc).
- Les entiers suivants sont le niveau de gris de chacun des pixels de l'image.

Version étendue. Ce format d'image est reconnu par tous les bons lecteurs d'images. Les logiciels de retouches d'images permettent d'exporter n'importe quelle image vers ce format. Par contre la structure du fichier peut être différente : tout d'abord il peut y avoir des lignes de commentaires (commençant par #), ensuite les valeurs de niveau de gris de chaque pixel peuvent être toutes sur une même ligne ou bien un seul pixel par ligne. C'est par exemple le cas du logiciel *gimp* qui produit des fichiers comme ci-dessous à gauche.

```
P2
# Commentaire !
5 4
255
0
255
64
255
0
255
...
```

```
P2
5 4
255
0 255 64 255 0 255 0 127 ...
```

Activité 2 (Lire et écrire des images).

Objectifs : convertir un fichier d'image en une matrice et inversement.

1. Programme une fonction `pgm_vers_matrice(fichier)` qui lit un fichier d'image au format « pgm » dont le nom est donné et renvoie une matrice.

Voici un exemple de fichier et la matrice associée :

P2

4 5

255

0 0 0 0

0 255 255 0

0 128 128 0

255 255 255 255

0 64 64 0

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 255 & 255 & 0 \\ 0 & 128 & 128 & 0 \\ 255 & 255 & 255 & 255 \\ 0 & 64 & 64 & 0 \end{pmatrix}$$

C'est préférable si ta fonction accepte les fichiers au format « pgm » étendu qui peuvent contenir des commentaires et où il n'y a qu'une nuance de gris par ligne. (Voir le cours ci-dessus.)

2. Fais le travail inverse en programmant une fonction `matrice_vers_pgm(M,fichier)` qui part d'une matrice M et écrit un fichier (dont le nom est donné) au format « pgm ».

Activité 3 (Convolution d'une image).

Objectifs : appliquer la convolution pour transformer une image.



Image originale



Flou



Bords



Piqué



Estampé

Programme une fonction `convolution_image(C,fichier_in,fichier_out)` qui à partir d'une matrice de convolution C de taille 3×3 et d'une image (dont le nom est donné) écrit les éléments d'une nouvelle image dans un fichier (dont le nom est aussi donné). Les fichiers d'images sont au format « pgm ».

C'est une fonction très simple qui utilise la fonction `convolution_entiere()`. Ensuite teste différentes convolutions !

Flou. La convolution avec la matrice suivante « floute » légèrement l'image. C'est parce que le niveau de gris de chaque nouveau pixel est la moyenne des niveaux des 9 pixels voisins de l'image de départ.

$$C = \begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}$$



Image originale



Flou

Essaye aussi le *flou gaussien* avec la matrice :

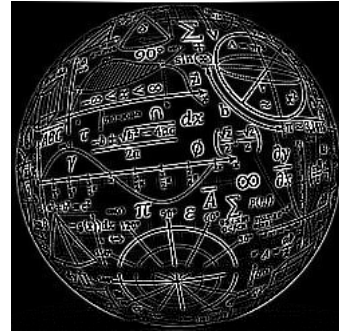
$$C = \begin{pmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{pmatrix}$$

Bords.

$$C = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$



Image originale



Bords

Essaie aussi les matrices :

$$C = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{ou} \quad C = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix}$$

Piqué.

C'est le contraire du flou !

$$C = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Voici un exemple avec l'image originale à gauche et l'image transformée à droite qui a l'air plus nette que l'originale !



Image originale



Piqué

Estampé.

$$C = \begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$



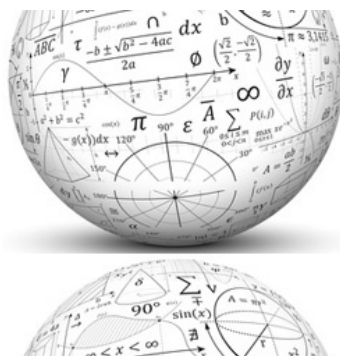
Image originale



Estampé

Décalage vers le haut.

Trouve la convolution qui décale l'image d'un pixel vers le haut et permet d'obtenir l'image suivante par itération.



Décalage itéré vers le haut

Cours 4 (Matrice de transformation).

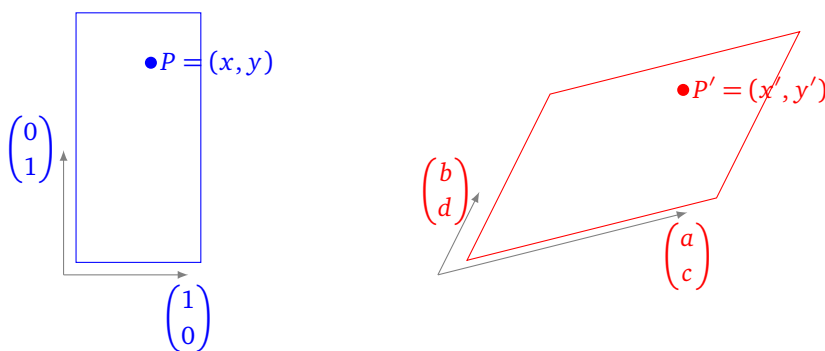
Voici de brefs rappels sur les matrices en se limitant aux matrices 2×2 . Pour l'interprétation géométrique, on identifie un point P de coordonnées (x, y) avec le vecteur $\vec{u} = \begin{pmatrix} x \\ y \end{pmatrix}$.

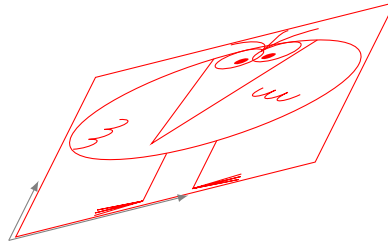
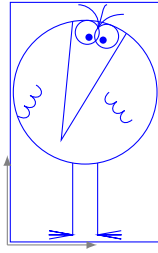
- **Multiplication par un vecteur.**

$$X = \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{et} \quad M = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \text{alors} \quad X' = AX = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

Autrement dit, les coordonnées (x', y') de l'image de $P = (x, y)$ sont données par :

$$\begin{cases} x' = ax + by \\ y' = cx + dy \end{cases}$$



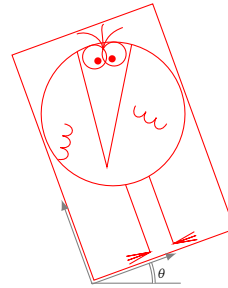
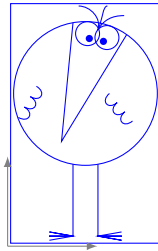


- **Matrice de rotation.**

$$T_{\theta} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$$

Autrement dit, pour $X' = T_{\theta}X$, on a :

$$\begin{cases} x' = x \cos \theta - y \sin \theta \\ y' = x \sin \theta + y \cos \theta \end{cases}$$



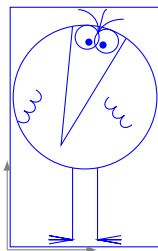
- **Dilatation.**

$$T = \begin{pmatrix} k_x & 0 \\ 0 & k_y \end{pmatrix}$$

Autrement dit :

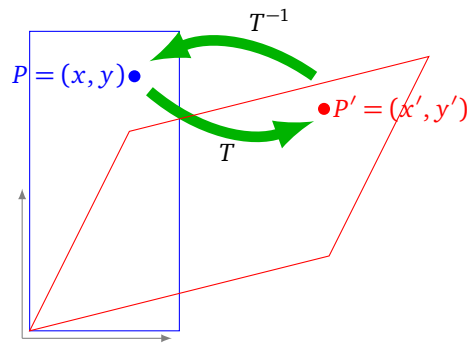
$$\begin{cases} x' = k_x \cdot x \\ y' = k_y \cdot y \end{cases}$$

Un exemple avec $k_x = 2$, $k_y = \frac{1}{2}$.



- **Inverse.** Une matrice M , dont le déterminant est non nul, admet un inverse M^{-1} :

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad M^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$



Activité 4 (Transformation d'images).

Objectifs : utiliser les matrices pour tourner, inverser ou déformer une image.

1. Dilatation.

Programme une fonction `dilatation_matrice(kx, ky, M)` qui effectue une dilatation de la matrice M . La matrice est agrandie d'un facteur k_x horizontalement et d'un facteur k_y verticalement en une nouvelle matrice N (ici k_x et k_y sont des entiers).

Exemple. Voici un exemple avec $k_x = 3$ et $k_y = 2$.

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad \text{devient} \quad N = \begin{pmatrix} 1 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 \\ 1 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 \\ 4 & 4 & 4 & 5 & 5 & 5 & 6 & 6 & 6 \\ 4 & 4 & 4 & 5 & 5 & 5 & 6 & 6 & 6 \\ 7 & 7 & 7 & 8 & 8 & 8 & 9 & 9 & 9 \\ 7 & 7 & 7 & 8 & 8 & 8 & 9 & 9 & 9 \end{pmatrix}$$

Formule. On obtient les coefficients de la nouvelle matrice N en fonction des coefficients de l'ancienne matrice M :

$$N[i][j] = M[i//k_y][j//k_x]$$

Indications.

- Commence par initialiser une matrice vide :

```
N = [[0 for j in range(kx*p)] for i in range(ky*n)]
```

de la bonne taille.

- Prends garde que horizontalement x correspond à j et verticalement y correspond à i .

Application. Dilate des images.



Image originale



Image dilatée ($k_x = 2, k_y = 3$)

L'image dilatée est plus grande mais sa résolution est plus faible, au final il n'y a ni gain ni perte d'information.

2. Matrice de transformation.

On souhaite généraliser la question précédente avec une transformation quelconque.

- Programme une fonction `vecteur_image(T, x, y)` qui à partir d'une matrice de transformation T et des coordonnées $\begin{pmatrix} x \\ y \end{pmatrix}$ d'un vecteur renvoie les coordonnées $\begin{pmatrix} x' \\ y' \end{pmatrix}$ de l'image de ce vecteur par T :

$$T = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = T \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

- Programme une fonction `inverse_matrice(T)` qui renvoie la matrice inverse de la matrice T (dont on suppose le déterminant non nul).

$$T = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad T^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

- Application.* Soit T_θ la matrice de rotation d'angle θ :

$$T_\theta = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$$

Soit $\theta = \frac{\pi}{3}$. Soit $P = (x, y) = (4, 5)$. Calcule l'image de P par la rotation d'angle θ . C'est le point $P' = (x', y')$ dont les coordonnées sont obtenues par multiplication de T_θ par le vecteur $\begin{pmatrix} x \\ y \end{pmatrix}$.

- Application.* Vérifie pour $\theta = \frac{\pi}{3}$ (par exemple) que la matrice de rotation $T_{-\theta}$ est égale à la matrice $(T_\theta)^{-1}$. (C'est juste une vérification par le calcul que l'opération inverse de tourner d'un angle θ c'est tourner d'un angle $-\theta$!)

3. Action de la transformation.

Le but est de programmer une fonction `transformation(T, M)` qui à partir d'une matrice T de taille 2×2 transforme une matrice M (de taille quelconque) en une nouvelle matrice N .

Application. Tu peux ensuite faire agir n'importe quelle matrice T pour transformer une image.

Dilatation (avec constantes quelconques).

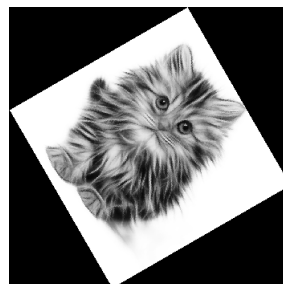
$$T = \begin{pmatrix} k_x & 0 \\ 0 & k_y \end{pmatrix}$$

avec $k_x = \pi = 3.14\dots$, $k_y = 1$



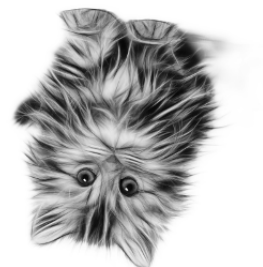
Rotation.

$$T = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \quad \text{avec } \theta = \frac{\pi}{3}$$



Symétrie.

$$T = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$



Transformation quelconque.

$$T = \begin{pmatrix} 3 & 1 \\ 1 & 5 \end{pmatrix}$$



Voici comment programmer cette fonction.

Première étape : taille de la nouvelle matrice.

On identifie la matrice M de taille $n \times p$ à une image rectangulaire. Les quatre coins de cette image ont pour coordonnées $P_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $P_1 = \begin{pmatrix} p \\ 0 \end{pmatrix}$, $P_2 = \begin{pmatrix} p \\ n \end{pmatrix}$, $P_3 = \begin{pmatrix} 0 \\ n \end{pmatrix}$. Calcule les coordonnées

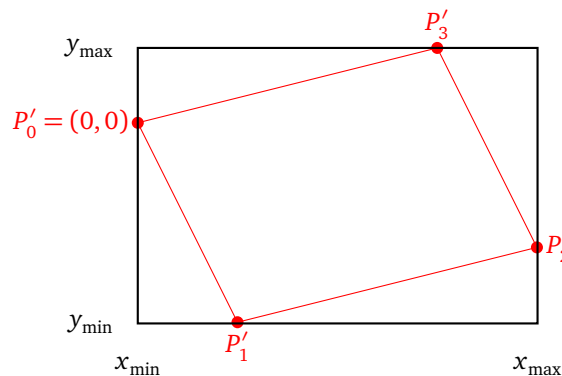
$P'_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix} = T \cdot P_i$ des coins transformés : ce sont les sommets d'un parallélogramme (ce n'est pas toujours un rectangle).

On note :

$$x_{\min} = \min(x_0, x_1, x_2, x_3) \quad \text{et} \quad x_{\max} = \max(x_0, x_1, x_2, x_3)$$

$$y_{\min} = \min(y_0, y_1, y_2, y_3) \quad \text{et} \quad y_{\max} = \max(y_0, y_1, y_2, y_3)$$

L'image transformée est incluse dans le rectangle défini par $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$!



On retient :

- le nombre de lignes de la matrice transformée est $n' = y_{\max} - y_{\min}$ (c'est la hauteur de l'image transformée),
- le nombre de colonnes de la matrice transformée est $p' = x_{\max} - x_{\min}$ (c'est la largeur de l'image transformée).

Seconde étape : formule de la transformation.

La nouvelle matrice N sera une matrice de taille $n' \times p'$.

Pour savoir combien vaut le coefficient $N[i'][j']$, il faut trouver le coefficient $M[i][j]$ correspondant dans la matrice de départ. (Autrement dit : la nouvelle l'image est de taille $n' \times p'$, le pixel (i', j') est de niveau de gris $N[i'][j']$ qui se calcule en fonction du niveau de gris $M[i][j]$.)

Il faut donc utiliser la matrice inverse T^{-1} . En plus il faut effectuer une translation de vecteur $\begin{pmatrix} x_{\min} \\ y_{\min} \end{pmatrix}$ pour recentrer l'origine. Ce qui donne :

$$\begin{pmatrix} j \\ i \end{pmatrix} = T^{-1} \begin{pmatrix} j' + x_{\min} \\ i' + y_{\min} \end{pmatrix}$$

et ensuite simplement :

$$N[i'][j'] = M[i][j]$$

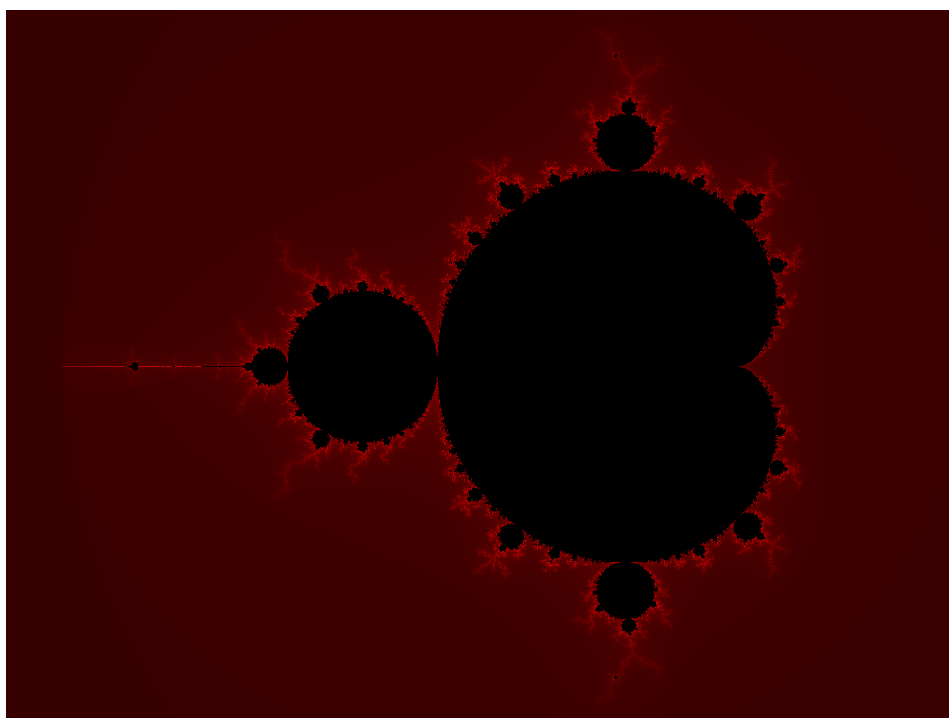
Exemple. Voici une dilatation avec des coefficients non entiers : le nombre de colonnes passe de 3 à 4 (facteur $k_x = \frac{4}{3}$), le nombre de lignes est doublé ($k_y = 2$).

$$T = \begin{pmatrix} \frac{4}{3} & 0 \\ 0 & 2 \end{pmatrix} \quad M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad N = \begin{pmatrix} 1 & 1 & 2 & 3 \\ 1 & 1 & 2 & 3 \\ 4 & 4 & 5 & 6 \\ 4 & 4 & 5 & 6 \\ 7 & 7 & 8 & 9 \\ 7 & 7 & 8 & 9 \end{pmatrix}$$

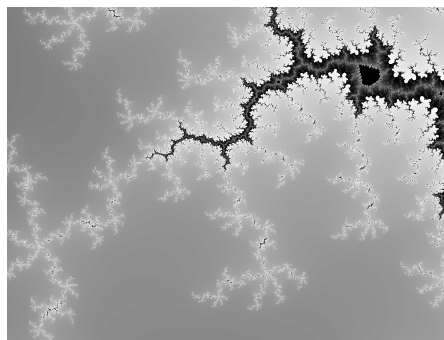
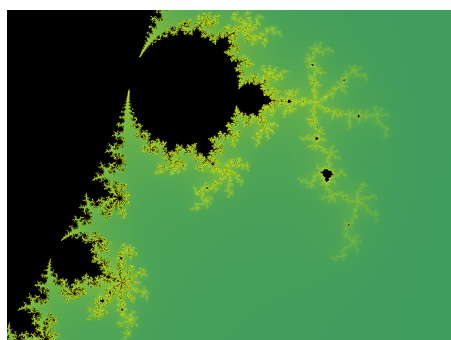
Tu peux maintenant transformer des images.

Ensemble de Mandelbrot

Tu vas découvrir un univers encore plus passionnant qu'Harry Potter : l'ensemble de Mandelbrot. C'est une fractale, c'est-à-dire que lorsque l'on zoome sur certaines parties de l'ensemble, on retrouve une image similaire à celle de départ. On découvrira aussi les ensembles de Julia.



Voici un zoom, puis un zoom du zoom ! Les points noirs forment l'ensemble de Mandelbrot.



Cette activité est proposée en deux versions : une avec les nombres complexes (pour ceux qui connaissent) et une version avec les nombres réels.

Cours 1 (L'ensemble de Mandelbrot (avec les nombres complexes)).

Voici la définition de l'ensemble de Mandelbrot pour ceux d'entre vous qui connaissent les nombres complexes.

On fixe un nombre complexe c . On définit une suite de nombres complexes par récurrence :

$$z_0 = 0 \quad \text{et pour } n \geq 0 \quad z_{n+1} = z_n^2 + c.$$

Si $|z_n|$ ne tend pas vers l'infini (lorsque $n \rightarrow +\infty$) alors c est par définition un point de l'ensemble de Mandelbrot $\mathcal{M}$.

Autrement dit, l'ensemble de Mandelbrot $\mathcal{M}$ est formé de toutes les valeurs $c \in \mathbb{C}$ telles que la suite récurrente (z_n) (qui dépend de c) reste bornée.

Activité 1 (Mandelbrot (version complexe)).

Objectifs : préparer le calcul de l'ensemble de Mandelbrot, en utilisant directement les nombres complexes.

1. Programme une fonction (toute simple) $f(z, c)$ qui pour z et c , des nombres complexes donnés, renvoie $z^2 + c$.
2. Programme une fonction `iterer(c)` qui pour un nombre complexe c donné, renvoie le nombre d'itérations qu'il a fallu pour que la suite (z_n) s'échappe « à l'infini » ; si au bout d'un certain nombre d'itérations la suite ne s'échappe pas, la fonction renvoie 0.

On peut prouver que (z_n) s'échappe vers l'infini dès que l'on trouve $i \geq 1$ tel que $|z_i| > 2$. On stoppe alors les calculs et on renvoie l'indice i . Si cela n'arrive pas au bout des 100 premiers termes par exemple, on considère que la suite ne s'échappera jamais et on arrête les calculs (on renvoie 0), et c est un point de l'ensemble de Mandelbrot.

Voici les détails afin de programmer cette fonction.

Algorithme.

- — Entrée : un nombre complexe c .
- — Sortie : le premier indice $i \geq 1$ avec $|z_i| > 2$, ou 0 si cela n'arrive pas pour les `MaxIter` premiers termes.
- Définir la constante `MaxIter`, par exemple `MaxIter = 100`.
- Poser $z = 0$ (correspondant à z_0).
- Poser $i = 1$.
- Tant que $|z| \leq 2$ et $i < \text{MaxIter}$:
 - Faire $z \leftarrow f(z, c)$.
 - Faire $i \leftarrow i + 1$.
- Une fois la boucle terminée, si $i = \text{MaxIter}$ alors renvoyer 0, sinon renvoyer i .

Tu peux passer directement à l'activité 3 pour afficher l'ensemble de Mandelbrot.

Cours 2 (L'ensemble de Mandelbrot (avec les nombres réels)).

Voici la définition de l'ensemble de Mandelbrot pour ceux qui ne connaissent pas encore les nombres complexes.

On fixe un point du plan (a, b) . On définit deux suites réelles par récurrence :

$$x_0 = 0 \quad \text{et} \quad y_0 = 0$$

et pour $n \geq 0$:

$$x_{n+1} = x_n^2 - y_n^2 + a \quad \text{et} \quad y_{n+1} = 2x_n y_n + b.$$

Si la suite des points (x_n, y_n) ne tend pas vers l'infini (lorsque $n \rightarrow +\infty$) alors (a, b) est par définition un point de l'ensemble de Mandelbrot $\mathcal{M}$.

Autrement dit, l'ensemble de Mandelbrot $\mathcal{M}$ est formé de tous les points $(a, b) \in \mathbb{R}^2$ tels que la suite des points (x_n, y_n) (qui dépend de (a, b)) reste bornée.

Activité 2 (Mandelbrot (version réelle)).

Objectifs : préparer le calcul de l'ensemble de Mandelbrot, en utilisant uniquement des nombres réels (les nombres complexes restent cachés).

1. Programme une fonction $f(x, y, a, b)$ qui pour x, y, a, b , des nombres réels, calcule

$$x' = x^2 - y^2 + a \quad \text{et} \quad y' = 2xy + b$$

et renvoie les deux nombres réels x' et y' .

2. Programme une fonction `iterer(a, b)` qui pour un couple de réel (a, b) donné, renvoie le nombre d'itérations qu'il a fallu pour que la suite (x_n, y_n) s'échappe « à l'infini » ; si au bout d'un certain nombre d'itérations la suite ne s'échappe pas, la fonction renvoie 0.

On peut prouver que (x_n, y_n) s'échappe vers l'infini dès que l'on trouve $i \geq 1$ tel que $x_i^2 + y_i^2 > 4$. On stoppe alors les calculs et on renvoie l'indice i . Si cela n'arrive pas au bout des 100 premiers termes par exemple, on considère que la suite ne s'échappera jamais et on arrête les calculs (on renvoie 0), et (a, b) est un point de l'ensemble de Mandelbrot.

Voici les détails afin de programmer cette fonction.

Algorithme.

- — Entrée : un couple de nombres réels (a, b) .
- — Sortie : le premier indice $i \geq 1$ avec $x_i^2 + y_i^2 > 4$, ou 0 si cela n'arrive pas pour les `MaxIter` premiers termes.
- Définir la constante `MaxIter`, par exemple `MaxIter = 100`.
- Poser $x = 0$ et $y = 0$ (correspondant à $(x_0, y_0) = (0, 0)$).
- Poser $i = 1$.
- Tant que $x^2 + y^2 \leq 4$ et $i < \text{MaxIter}$:
 - Faire $x, y \leftarrow f(x, y, a, b)$.
 - Faire $i \leftarrow i + 1$.
- Une fois la boucle terminée, si $i = \text{MaxIter}$ alors renvoyer 0, sinon renvoyer i .

Dans les activités suivantes tu vas afficher l'ensemble de Mandelbrot.

Activité 3 (Préparer l'affichage de l'ensemble de Mandelbrot).

Objectifs : se préparer à afficher l'ensemble de Mandelbrot. Le tout en couleur !

On utilise le module `tkinter` pour afficher la fractale.

Le code principal sera le suivant (la fonction `mandelbrot()` sera définie dans l'activité suivante) :

```

from tkinter import *
root = Tk()
canvas = Canvas(root, width=Nx, height=Ny, background="white")
canvas.pack(side=LEFT, padx=5, pady=5)

mandelbrot()

root.mainloop()

```

1. Programme une fonction `afficher_pixel(i, j, couleur)` qui affiche un pixel coloré en position (i, j) d'une fenêtre graphique (qui sera définie après).

Tu peux par exemple tracer un segment à l'aide de `create_line()` de (i, j) à $(i + 1, j)$.

2. Programme une fonction `choix_couleur(i)` qui renvoie une couleur en fonction du nombre i (qui correspondra au nombre d'itérations).

```

if i == 0:
    R, V, B = 0, 0, 0
else:
    R, V, B = 50 + 2*i, 0, 0
couleur = '%02x%02x%02x' % (R, V, B)

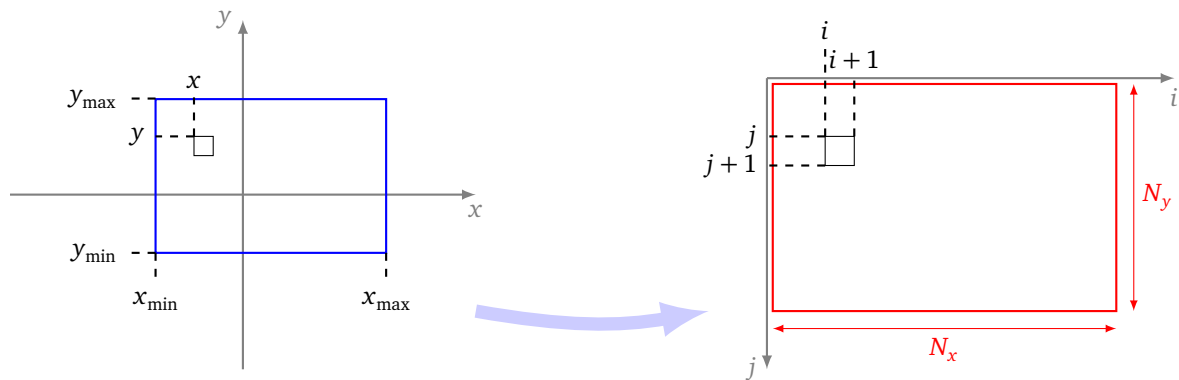
```

3. Définis des variables `xmin`, `xmax`, `ymin`, `ymax` qui correspondent à la zone de l'ensemble de Mandelbrot à visualiser. Par exemple pour avoir tout l'ensemble de Mandelbrot on fait :

```

xmin, xmax = -2.2, 1
ymin, ymax = -1.2, 1.2

```



Plan des coordonnées (x, y)

Écran des coordonnées (i, j)

Le plan de coordonnées (x, y) est représenté sur la figure de gauche ; le grand rectangle correspond à la zone à afficher ; le petit carré est la zone qui sera représentée par un pixel. Sur la figure de droite c'est l'écran ; les coordonnées sont données par des coordonnées entières (i, j) ; l'écran est composé de pixels (un seul est dessiné), chacun de largeur 1.

4. Définis deux variables `Nx` et `Ny` qui correspondent à la taille de la fenêtre graphique à afficher. Par exemple pour une fenêtre de largeur `Nx = 400` pixels et de hauteur correspondant à la zone à visualiser, faire :

```

Nx = 400
Ny = round( (ymax-ymin)/(xmax-xmin) * Nx )

```

Activité 4 (Tracer l'ensemble de Mandelbrot).

Objectifs : dessiner l'ensemble du Mandelbrot ou un zoom.

- Tu as une zone rectangulaire définie par $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$ qui correspond à la zone de l'ensemble de Mandelbrot à visualiser.
- Cette zone s'affichera dans une fenêtre de largeur N_x pixels et de hauteur N_y pixels.
- Définis d'abord deux variables pas_x , pas_y qui correspondent à la taille couverte par un pixel selon les formules :

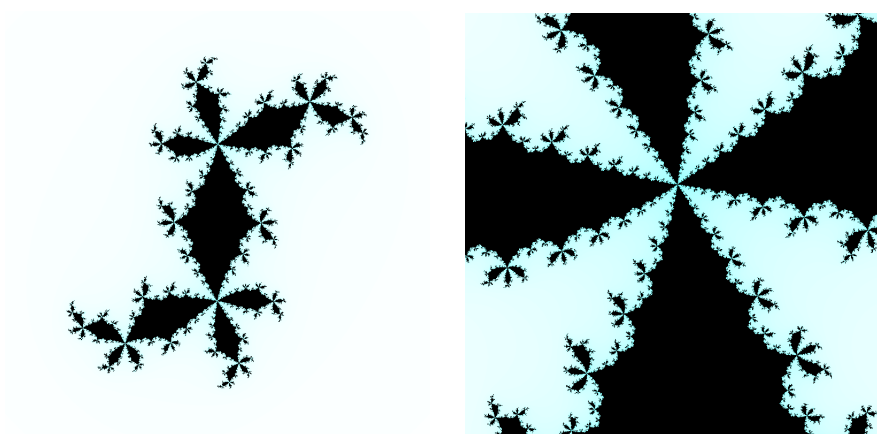
$$pas_x = (x_{\max} - x_{\min}) / N_x \qquad pas_y = (y_{\max} - y_{\min}) / N_y$$

Programme une fonction `mandelbrot()` qui trace la zone désirée de l'ensemble de Mandelbrot selon les instructions suivantes :

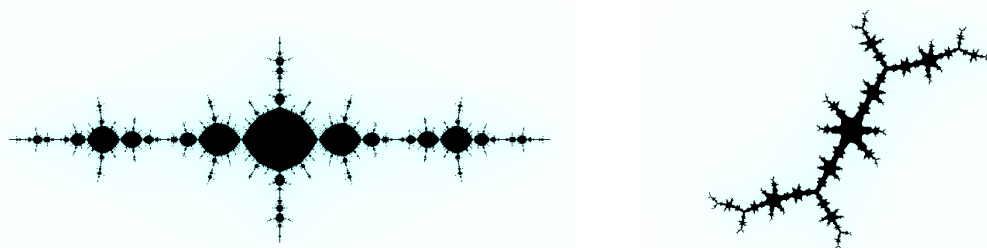
- Partir de $a = x_{\min}$ et $b = y_{\min}$.
- Pour i variant de 0 à N_x :
 - Pour j variant de 0 à N_y :
 - ★ (Version complexe) Poser $c = a + b \cdot 1j$ (c'est-à-dire $c = a + ib \in \mathbb{C}$) et calculer `vitesse = iterer(c)`.
 - ★ (Version réelle) Calculer `vitesse = iterer(a,b)`.
 - ★ Choisir une couleur correspondant à la vitesse d'échappement `couleur = choix_couleur(vitesse)`.
 - ★ Allumer le pixel correspondant par `afficher_pixel(i,j,couleur)`.
 - ★ Faire $b = b + pas_y$.
 - Faire $b = y_{\min}$.
 - Faire $a = a + pas_x$.

Le jeu est maintenant double : trouver les plus jolies couleurs possibles (en modifiant la fonction `choix_couleur()`) et surtout chercher des zooms avec les plus belles formes possibles.

Les ensembles de Julia se construisent par une méthode proche de celle utilisée pour l'ensemble de Mandelbrot. Cependant les images obtenues sont très différentes et en plus il y a une infinité d'ensembles de Julia possibles.



Ci-dessus l'ensemble de Julia avec $c = 0.3 + 0.55i$ ($a = 0.3$, $b = 0.55$), appelé le « lapin de Douady », avec à droite un zoom. Ci-dessous l'ensemble de Julia pour $c = -1.31$ (à gauche) et $c = -0.101 + 0.956i$ (à droite).



Cours 3 (Les ensembles de Julia (avec les nombres complexes)).

Voici la définition des ensembles de Julia pour ceux d'entre vous qui connaissent les nombres complexes.

On fixe un nombre complexe c . On va définir l'ensemble de Julia $\mathcal{J}(c)$ associé à cette valeur. Pour chaque $z_0 \in \mathbb{C}$ on définit une suite de nombres complexes par récurrence pour $n \geq 0$:

$$z_{n+1} = z_n^2 + c$$

Si $|z_n|$ ne tend pas vers l'infini (lorsque $n \rightarrow +\infty$) alors z_0 est par définition un point de l'ensemble de Julia $\mathcal{J}(c)$. Cette fois l'ensemble de Julia $\mathcal{J}(c)$ est formé de toutes les valeurs $z_0 \in \mathbb{C}$ telles que la suite récurrente (z_n) (qui dépend de z_0) reste bornée.

La différence avec la définition de l'ensemble de Mandelbrot est qu'ici, le terme c est fixé pour chaque fractale, et c'est le terme initial z_0 que l'on fait varier.

Activité 5 (Julia (version complexe)).

Objectifs : dessiner des ensembles de Julia, en utilisant les nombres complexes.

Il s'agit d'adapter ton programme qui a servi à dessiner l'ensemble de Mandelbrot.

- Conserve ta fonction $f(z, c)$ qui pour z et c , des nombres complexes donnés, renvoie $z^2 + c$.
- Adapte la fonction `iterer()` en une fonction `iterer(z0, c)` pour tenir compte du terme initial z_0 .
- Pour un complexe c fixé, programme une fonction `julia(c)` qui affiche l'ensemble de Julia $\mathcal{J}(c)$. C'est presque comme pour l'ensemble de Mandelbrot, mais cette fois lorsque tu fais varier a, b c'est pour définir $z_0 = a + ib$; c lui reste fixé.

Cours 4 (Les ensembles de Julia (version réelle)).

Voici la définition des ensembles de Julia en utilisant seulement les nombres réels.

On fixe un couple de réels (a, b) . On va définir l'ensemble de Julia $\mathcal{J}(a, b)$ associé à ce couple. Pour chaque couple $(x_0, y_0) \in \mathbb{R}^2$ on définit une suite de nombres réels par récurrence pour $n \geq 0$:

$$x_{n+1} = x_n^2 - y_n^2 + a \quad \text{et} \quad y_{n+1} = 2x_n y_n + b$$

Si la suite des points (x_n, y_n) ne tend pas vers l'infini (lorsque $n \rightarrow +\infty$) alors (x_0, y_0) est par définition un point de l'ensemble de Julia $\mathcal{J}(a, b)$. Cette fois l'ensemble de Julia $\mathcal{J}(a, b)$ est formé de tous les couples $(x_0, y_0) \in \mathbb{R}^2$ tels que la suite (x_n, y_n) (qui dépend de (x_0, y_0)) reste bornée.

La différence avec la définition de l'ensemble de Mandelbrot est qu'ici, le couple (a, b) est fixé pour chaque fractale, et c'est le couple initial (x_0, y_0) que l'on fait varier.

Activité 6 (Julia (version réelle)).

Objectifs : dessiner des ensembles de Julia, en utilisant les nombres réels.

Il s'agit d'adapter ton programme qui a servi à dessiner l'ensemble de Mandelbrot.

- Conserve ta fonction $f(x, y, a, b)$ qui pour x, y et a, b , nombres réels donnés, renvoie les deux réels $x' = x^2 - y^2 + a$ et $y' = 2xy + b$.
- Adapte ta fonction `iterer()` en une fonction `iterer(x0, y0, a, b)` pour tenir compte des termes initiaux x_0 et y_0 .
- Pour un couple (a, b) fixé, programme une fonction `julia(a, b)` qui affiche l'ensemble de Julia $\mathcal{J}(a, b)$. C'est presque comme pour l'ensemble de Mandelbrot, mais cette fois tu fais varier x_0 et y_0 à chaque pixel de l'écran ; a et b eux restent fixés.

Cours 5 (Pour quelques secondes de moins...).

Les calculs pour tracer l'ensemble de Mandelbrot (et ceux de Julia) sont très longs. Il n'y a pas de solutions immédiates avec Python pour aller plus vite. (Les scripts Python sont interprétés et sont plus longs à exécuter qu'un programme compilé.)

Voici quelques petites astuces pour aller plus vite.

- Pour calculer x^2 il peut être plus rapide de calculer `x*x` que `x**2`.
- Il est plus rapide de vérifier $x^2 + y^2 > 4$ que $\sqrt{x^2 + y^2} > 2$.
- Enfin il est important de ne pas recalculer plusieurs fois la même expression. Par exemple à chaque itération, on a besoin de calculer

$$x' = x^2 - y^2 + a \quad \text{et} \quad y' = 2xy + b$$

et de vérifier si

$$x^2 + y^2 > 4.$$

Donc on calcule deux fois x^2 et deux fois y^2 . Pour éviter cela on peut donc commencer par calculer $x_2 = x^2$, $y_2 = y^2$, puis faire les calculs :

$$x' = x_2 - y_2 + a, \quad y' = 2xy + b \quad \text{et} \quad x_2 + y_2 > 4.$$

On est donc passé du calcul de 5 multiplications (4 carrés et le produit xy) à 3 multiplications (2 carrés et le produit xy).

- Enfin, on peut éviter de calculer des produits de deux nombres différents mais uniquement des carrés (en théorie c'est un tout petit peu plus rapide). Il suffit de remarquer que

$$2xy = (x + y)^2 - x^2 - y^2$$

(il n'y a qu'un nouveau carré à calculer car on a déjà calculé x^2 et y^2). Au total on a donc seulement 3 carrés à calculer à chaque itération.

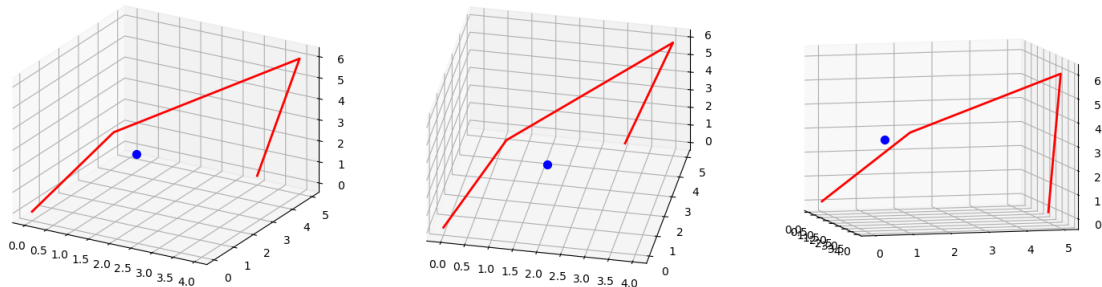
Comment dessiner des objets dans l'espace et comment les représenter sur un plan ?

Cours 1 (Images 3D avec matplotlib).

Avec le module matplotlib il est assez facile de tracer une représentation des objets dans l'espace. Le principe est similaire à l'affichage dans le plan, sauf bien sûr qu'il faut préciser trois coordonnées x, y, z pour déterminer un point de l'espace.

Voici un code très simple qui affiche :

- un point bleu de coordonnées (2, 1, 3),
- des segments rouges qui relient les points de la liste (0, 0, 0), (1, 2, 3), (4, 5, 6), (3, 5, 0).



Une fenêtre s'affiche dans laquelle sont dessinés le point et les segments ainsi que les plans quadrillés de coordonnées. L'image est dynamique : à l'aide de la souris tu peux faire tourner le dessin afin de changer de point de vue.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Initialisation
fig = plt.figure()
ax = fig.gca(projection='3d', proj_type = 'ortho')

# Affichage d'un point
x,y,z = (2,1,3)
ax.scatter(x,y,z,color='blue',s=50)

# Segments reliant des points
points = [(0,0,0),(1,2,3),(4,5,6),(3,5,0)]
```

```

liste_x = [x for x,y,z in points]
liste_y = [y for x,y,z in points]
liste_z = [z for x,y,z in points]

ax.plot(liste_x,liste_y,liste_z,color='red',linewidth=2)

# Affichage
plt.show()

```

Avertissement. Pour afficher des segments la commande plot n'est pas très naturelle (mais c'était déjà le cas dans le plan). Par exemple pour relier le point (1, 2, 3) au point (4, 5, 6) on donne d'abord la liste des x , puis la liste des y , puis la liste des z :

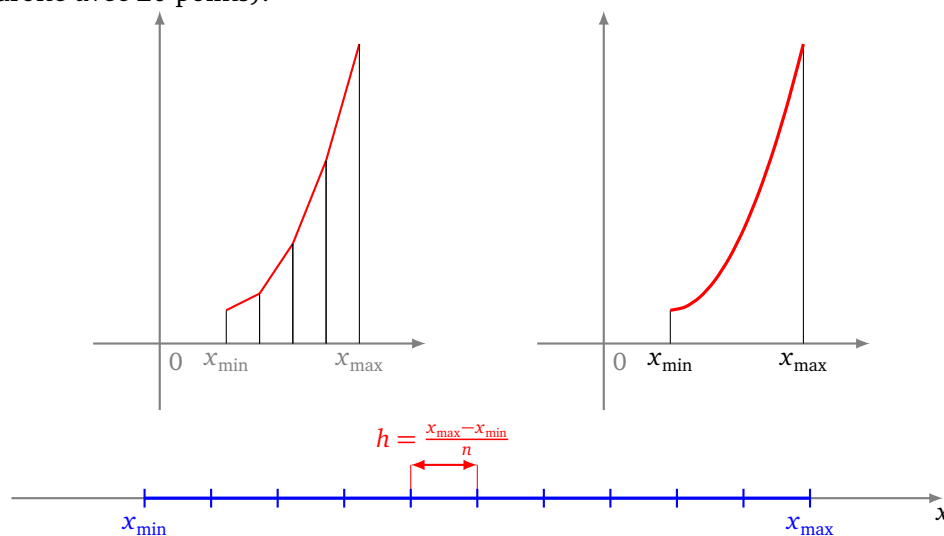
```
plot([1,4],[2,5],[3,6])
```

Cours 2 (Surface d'équation $z = f(x, y)$).

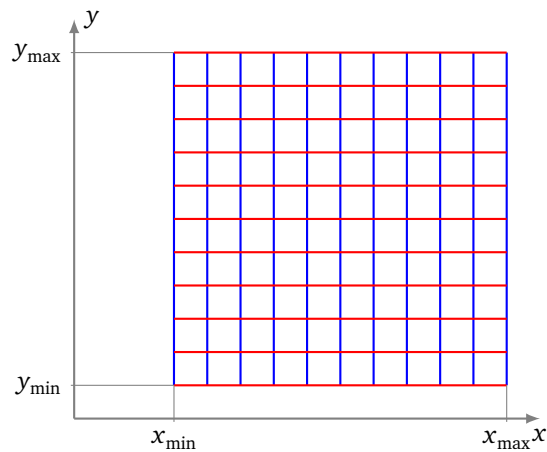
- Une **fonction de deux variables** associe à un couple de réels (x, y) un réel $f(x, y)$, c'est donc une fonction $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, $(x, y) \mapsto f(x, y)$.
- Le **graphe** d'une fonction de deux variables est la surface d'équation $z = f(x, y)$, autrement dit c'est

$$\{(x, y, z) \in \mathbb{R}^3 \mid z = f(x, y)\}$$

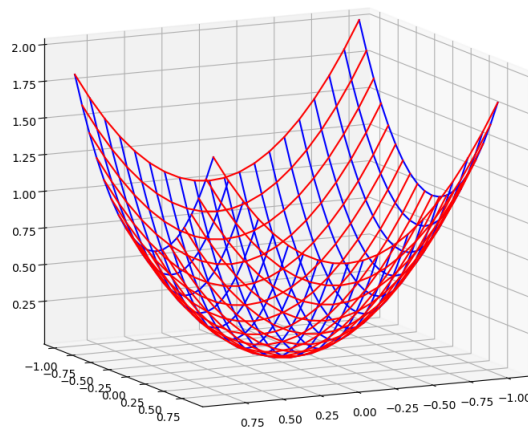
- Pour tracer le graphe d'une fonction d'une seule variable, on relie des points $(x, f(x))$ entre eux (sur la figure de gauche 5 points), si les points sont suffisamment proches, la courbe a l'air lisse (sur la figure de droite avec 20 points).



- Pour tracer une surface associée à une fonction de deux variables on commence par quadriller le plan (x, y) .

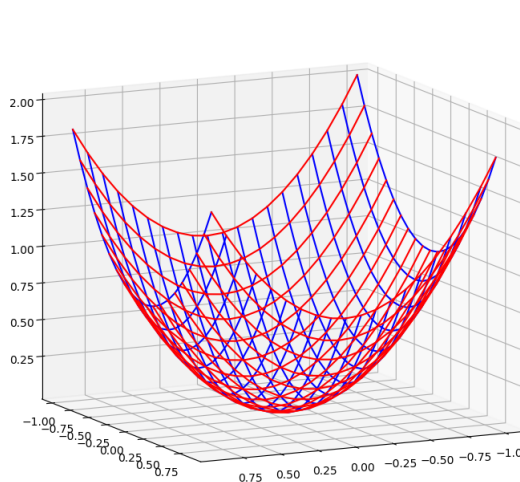


- Ensuite on trace les points $(x, y, f(x, y))$ au-dessus de chaque ligne verticale bleue du plan en les reliant puis on fait la même chose sur chaque ligne horizontale rouge.

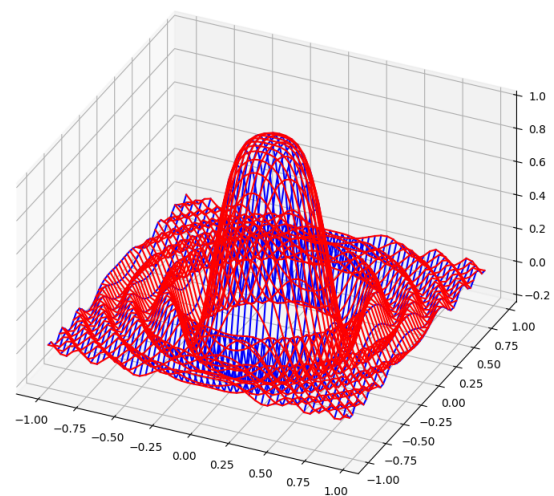


Activité 1 (Surfaces).

Objectifs : tracer la surface d'équation $z = f(x, y)$ donnée par une fonction de deux variables.



Bol – $f(x, y) = x^2 + y^2$

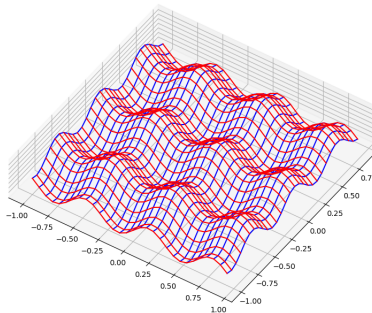


Goutte qui tombe dans l'eau –
 $f(x, y) = \frac{\sin(r)}{r}$ où $r = 20(x^2 + y^2)$

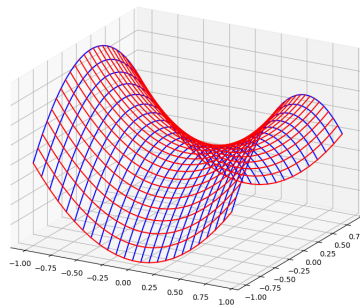
1. Programme une fonction $f(x, y)$ qui renvoie une valeur en fonction de x et de y . Voici des exemples de fonctions :

- $f(x, y) = x^2 + y^2$ (un bol).
- $f(x, y) = \frac{\sin(r)}{r}$ où $r = 20(x^2 + y^2)$ (une goutte qui tombe dans l'eau).
- $f(x, y) = \sin(10x) + \cos(10y)$ (une boîte d'œufs).
- $f(x, y) = x^2 - y^2$ (une selle de cheval).
- $f(x, y) = \exp(-\frac{1}{3}x^3 + x - y^2)$ pour $x \in [-2, 3]$ et $y \in [-2.5, 2.5]$ (un sommet et un col).

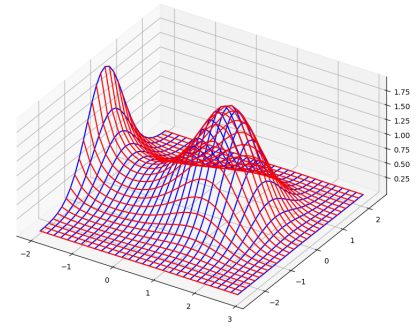
Les graphes d'équation $z = f(x, y)$ sont dessinés ci-dessus ou ci-dessous. À l'exception de la dernière fonction, les tracés sont obtenus pour $x \in [-1, +1]$ et $y \in [-1, +1]$.



Boîte d'œufs –
 $f(x, y) = \sin(10x) + \cos(10y)$



Selle de cheval –
 $f(x, y) = x^2 - y^2$



Sommet et col –
 $f(x, y) = \exp(-\frac{1}{3}x^3 + x - y^2)$
pour $x \in [-2, 3]$ et
 $y \in [-2.5, 2.5]$

2. (a) Définis des constantes globales $x_{\min} = -1$, $x_{\max} = +1$ pour définir l'intervalle $x \in [-1, +1]$ et $y_{\min} = -1$, $y_{\max} = +1$ pour définir l'intervalle $y \in [-1, +1]$.

Définis aussi une constante globale $nbpoints = 10$ qui correspond au nombre N de découpages. Chaque ligne est formée de $N + 1$ points; il y a aussi en tout $N + 1$ lignes tracées dans chaque direction.

(b) Programme une fonction `liste_points_xcst(x)` qui renvoie une liste de points (x, y, z) où :

- x est la valeur donnée en paramètre de la fonction,
- y prend les valeurs $y_{\min} + kh$ pour $k = 0, 1, \dots, N$ et $h = \frac{y_{\max} - y_{\min}}{N}$ (on rappelle que $N = nbpoints$),
- $z = f(x, y)$.

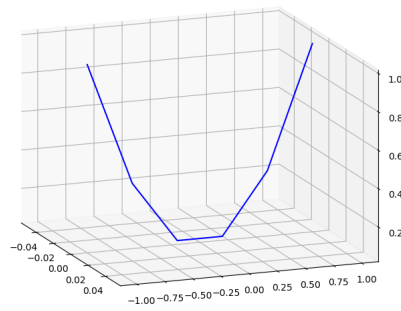
Par exemple pour la fonction $f(x, y) = x^2 + y^2$, $x = 0$ et l'intervalle $[-1, +1]$ des y découpé en $N = 5$ morceaux, la liste des points renvoyée est une liste de $N + 1 = 6$ points :

$[(0, -1, 1), (0, -0.6, 0.36), (0, -0.2, 0.04),$
 $(0, 0.2, 0.04), (0, 0.6, 0.36), (0, 1, 1)]$

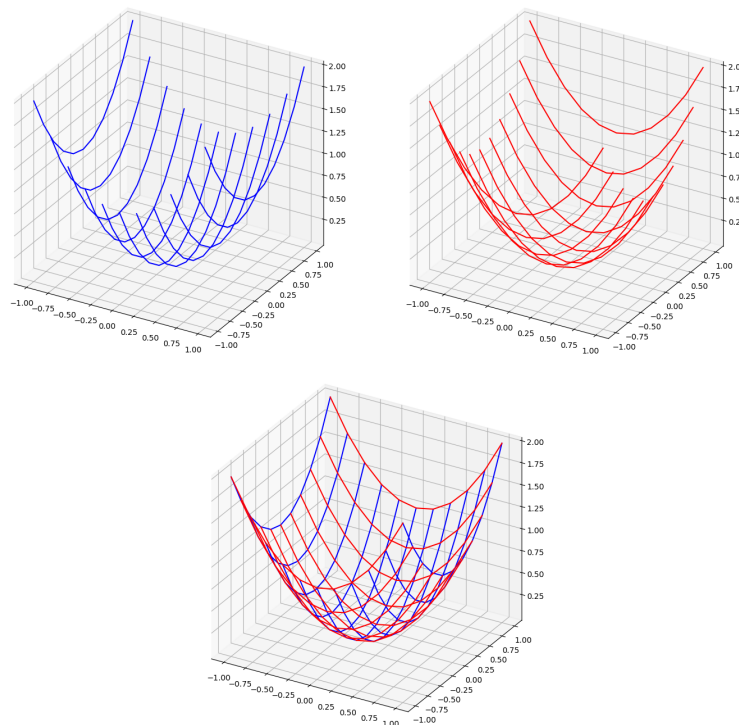
(c) Fais le même travail pour une fonction `liste_points_ycst(y)` où cette fois y est fixé et c'est x qui varie.

3. (a) Programme une fonction `trace_ligne(liste_points)` (ou mieux `trace_ligne(liste_points, couleur='gray')` permettant de changer la couleur du trait) qui relie les points (x, y, z) de la liste par des segments (du premier au dernier).

Voici l'affichage des 6 points de la liste de la question précédente.

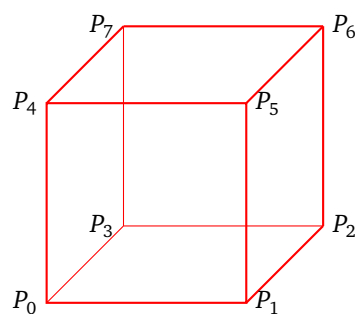


- (b) Programme ensuite une fonction `trace_surface()` qui dessine la surface en traçant $N + 1$ lignes (bleues) pour chacune desquelles x est constant (figure de gauche) et aussi $N + 1$ lignes (rouges) pour chacune desquelles y est constant (figure du milieu) pour obtenir une représentation de la surface (figure de droite).



Cours 3 (Perspective).

Ceci n'est pas un cube !

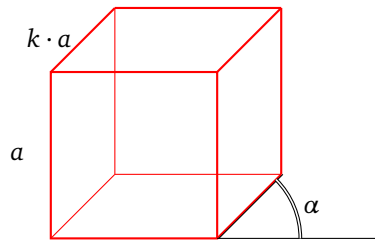


C'est juste une union de segments du plan. Notre cerveau est capable de reconstruire un objet en trois dimensions à partir d'un image plane. Pour dessiner sur une feuille un objet 3D il faut donc une formule

qui transforme un point (x, y, z) de l'espace en un point (X, Y) du plan. Il existe différentes formules, en voici quelques unes. Pour les dessins voir l'activité qui suit.

Perspective cavalière.

Elle est définie par une constante k qui réduit les longueurs des segments obliques et un angle α .



$$\begin{cases} X = x + k \cos(\alpha)y \\ Y = z + k \sin(\alpha)y \end{cases}$$

Les constantes α et k sont le plus souvent $(\alpha = \frac{\pi}{4}, k = \frac{1}{2})$ ou bien $(\alpha = \frac{\pi}{6}, k = 0.7)$.

Perspective axonométrique.

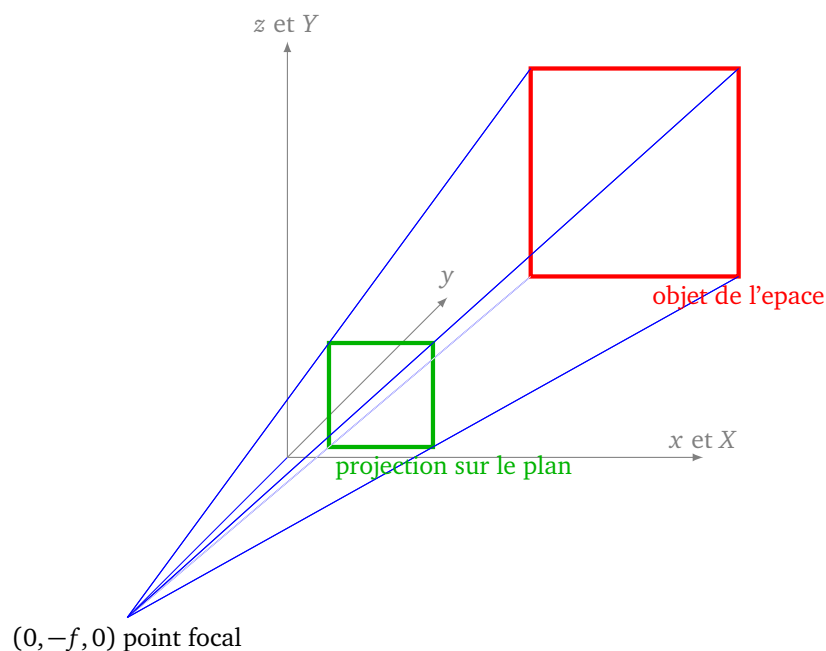
Cette opération consiste à tourner d'abord l'objet selon deux axes (il y a donc deux angles ω et α), avant de projeter sur un plan.

$$\begin{cases} X = \cos(\omega)x - \sin(\omega)y \\ Y = -\sin(\omega)\sin(\alpha)x - \cos(\omega)\sin(\alpha)y + \cos(\alpha)z \end{cases}$$

Où ω et α sont des angles fixés. Dans le cas particulier $\omega = 0.61$ et $\alpha = \frac{\pi}{4}$ on obtient la *perspective isométrique*.

Perspective conique.

Il s'agit de regarder un objet de l'espace depuis un point $(0, -f, 0)$ et de le projeter sur le plan $(y = 0)$; f est une valeur constante qui s'appelle la *focale*.



Les formules sont :

$$\begin{cases} X = kx \\ Y = kz \end{cases} \quad \text{avec} \quad k = \frac{f}{y+f}.$$



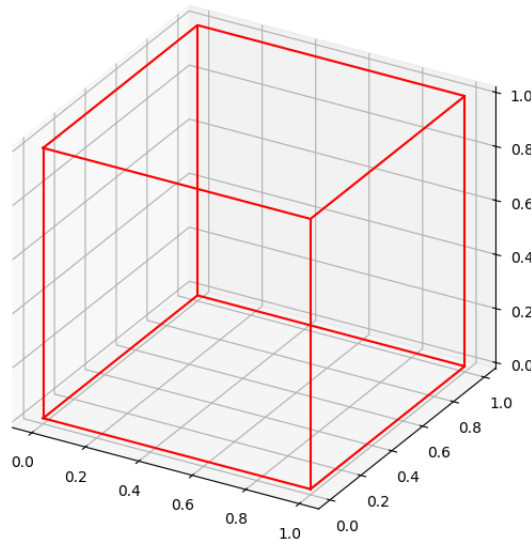
Activité 2 (Perspective).

Objectifs : transformer un objet de l'espace en un objet du plan afin de l'afficher.

Pour les exemples on va afficher différentes projections du cube donné par les coordonnées suivantes :

`cube = [(0,0,0), (1,0,0), (1,1,0), (0,1,0), (0,0,1), (1,0,1), (1,1,1), (0,1,1)]`

1. Programme une fonction `affiche_cube_3d(cube)` qui à partir d'une liste de 8 points $[P_0, P_1, \dots, P_7]$ trace l'affichage 3D du cube $(P_0, P_1, \dots, P_7)$. Il s'agit juste de tracer les 12 arêtes du cube !



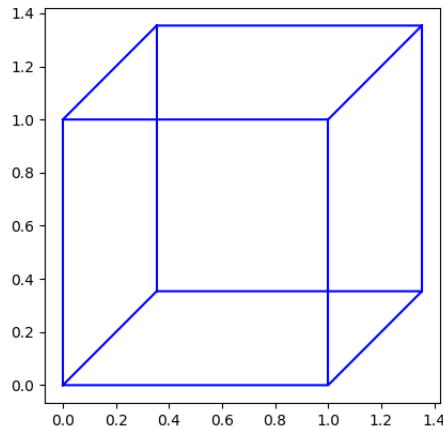
2. Perspective cavalière.

- (a) Programme une fonction `perspective_cavaliere(P)` (ou mieux `perspective_cavaliere(P, alpha=pi/4, k=0.5)`) qui à partir d'un point P de l'espace de coordonnées (x, y, z) renvoie le point Q du plan de coordonnées (X, Y) selon la formule :

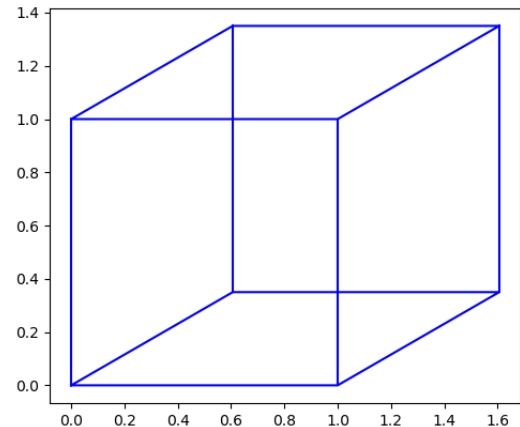
$$\begin{cases} X &= x + k \cos(\alpha)y \\ Y &= z + k \sin(\alpha)y \end{cases}$$

Pour les constantes α et k , on testera $(\alpha = \frac{\pi}{4}, k = \frac{1}{2})$ puis $(\alpha = \frac{\pi}{6}, k = 0.7)$.

- (b) Programme une fonction `affiche_cube_2d(cube2d)` qui à partir d'une liste $[Q_0, Q_1, \dots, Q_7]$ de 8 **points du plan**, relie les points deux à deux comme si c'était les arêtes d'un cube.
- (c) À partir des sommets du cube 3D $(P_0, P_1, \dots, P_7)$, calcule sa projection $(Q_0, Q_1, \dots, Q_7)$ dans le plan et affiche cette projection du cube. Voici le résultat ci-dessous : il s'agit bien ici de deux images du plan !



$$\alpha = \frac{\pi}{4}, k = \frac{1}{2}$$

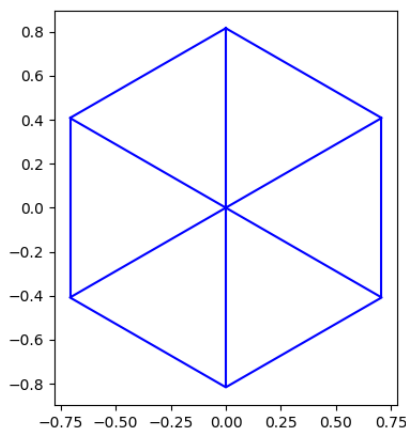


$$\alpha = \frac{\pi}{6}, k = 0.7$$

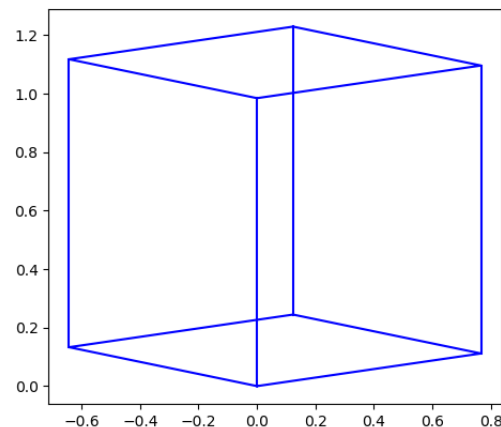
3. **Perspective axonométrique.** Fais le même travail pour une fonction `perspective_axonometrique(P)` (ou mieux `perspective_axonometrique(P, alpha=0.61, omega=pi/4)`) qui pour $P = (x, y, z)$ point de l'espace renvoie le point $Q = (X, Y)$ du plan suivant les formules :

$$\begin{cases} X = \cos(\omega)x - \sin(\omega)y \\ Y = -\sin(\omega)\sin(\alpha)x - \cos(\omega)\sin(\alpha)y + \cos(\alpha)z \end{cases}$$

Affiche la projection du cube.



Perspective isométrique : $\omega = 0.61, \alpha = \frac{\pi}{4}$



Perspective axonométrique avec $\omega = 30^\circ$,
 $\alpha = -10^\circ$ (à convertir en radians)

Avec la perspective isométrique (à gauche) toutes les arêtes projetées ont la même longueur. Ici la projection n'est pas très lisible car deux sommets sont projetés sur le même point.

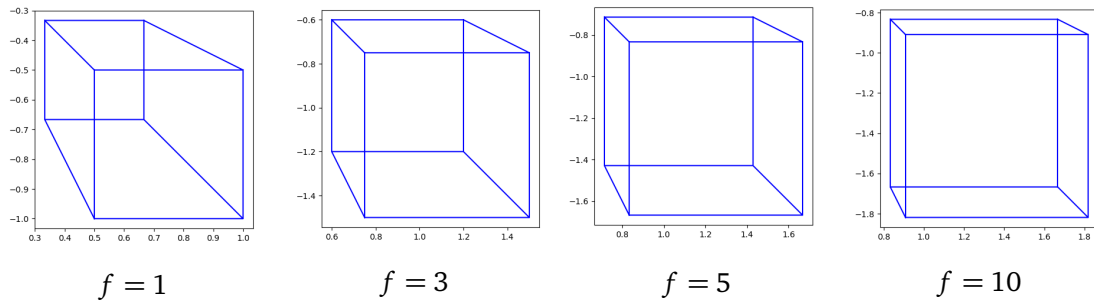
4. **Perspective conique.** Fais le même travail pour une fonction `perspective_conique(P)` (ou mieux `perspective_conique(P, f=2)`) qui pour $P = (x, y, z)$ point de l'espace renvoie le point $Q = (X, Y)$ du plan suivant les formules :

$$\begin{cases} X = kx \\ Y = kz \end{cases} \quad \text{avec} \quad k = \frac{f}{y+f}.$$

Voici l'affichage du cube :

```
cube = [(1,1,-1), (2,1,-1), (2,2,-1), (1,2,-1), (1,1,-2), (2,1,-2),
        (2,2,-2), (1,2,-2)]
```

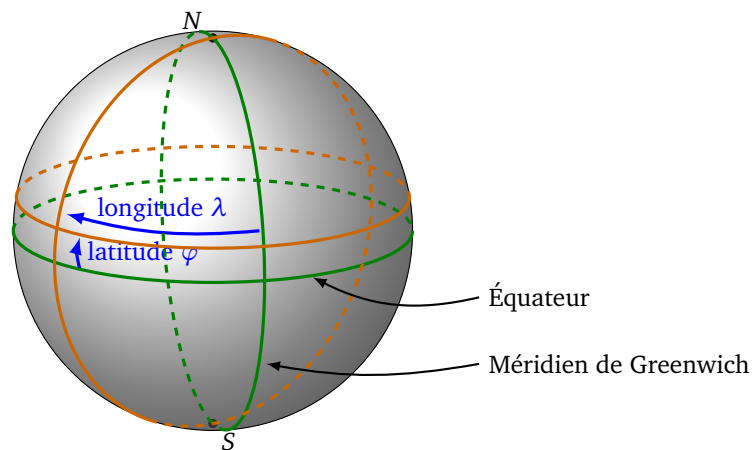
pour différentes valeurs de la focale f .



Cours 4 (Coordonnées sur la sphère : latitude et longitude).

Pour se repérer dans l'espace on peut utiliser les coordonnées (x, y, z) d'un repère orthonormé direct. Mais pour se repérer à la surface de la Terre ou plus généralement sur une sphère, on peut aussi utiliser les **coordonnées sphériques** $[r : \varphi : \lambda]$ où :

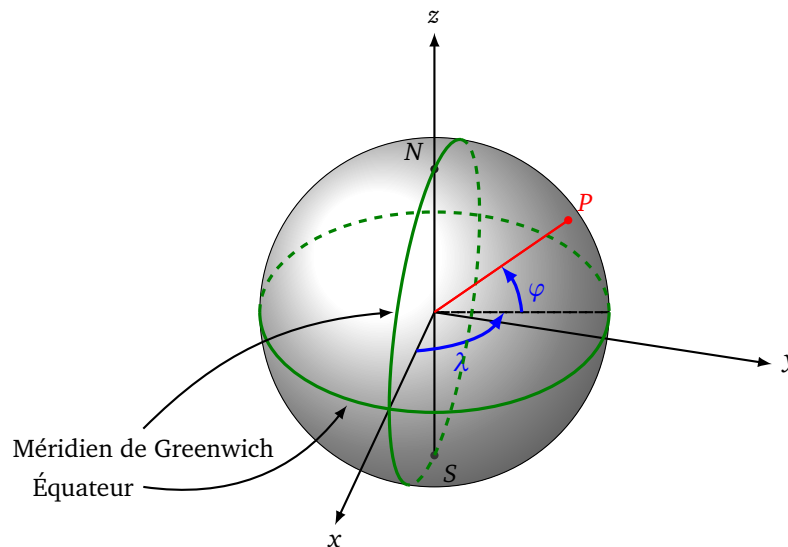
- $r > 0$ est le rayon de la sphère,
- φ est la **latitude**, c'est un angle de $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ (autrement dit compris entre -90° et $+90^\circ$),
- λ est la **longitude**, c'est un angle de $]-\pi, +\pi]$ (autrement dit compris entre -180° et $+180^\circ$).



Passage vers les coordonnées cartésiennes.

On trouve (x, y, z) en fonction de $[r : \varphi : \lambda]$ par les formules suivantes :

$$\begin{cases} x = r \cos \varphi \cos \lambda \\ y = r \cos \varphi \sin \lambda \\ z = r \sin \varphi \end{cases}$$



Passage vers les coordonnées sphériques.

Pour trouver $[r : \varphi : \lambda]$ à partir de (x, y, z) c'est un peu plus compliqué.

Le rayon

$$r = \sqrt{x^2 + y^2 + z^2}$$

La latitude

$$\varphi = \arcsin\left(\frac{z}{r}\right)$$

La longitude

$$\lambda = \arcsin\left(\frac{1}{\cos \varphi} \frac{y}{r}\right)$$

Activité 3 (Coordonnées sur la sphère : latitude et longitude).

Objectifs : se repérer sur la sphère grâce à la latitude et la longitude.

1. Vers les coordonnées cartésiennes.

Programme une fonction `latlong_vers_xyz(r, phi, lamb)` qui renvoie les coordonnées cartésiennes (x, y, z) du point de coordonnées sphériques $[r : \varphi : \lambda]$.

Attention. Ne pas utiliser une variable nommée `lambda` qui est un nom réservé par Python pour autre chose.

Question. Quelles sont les coordonnées (x, y, z) du point vérifiant $r = 1$, de latitude $\varphi = 45^\circ$ et de longitude $\lambda = 30^\circ$. (N'oublie pas de convertir les degrés en radians.)

2. Vers les coordonnées sphériques.

Programme une fonction `xyz_vers_latlong(x, y, z)` qui renvoie les coordonnées sphériques $[r : \varphi : \lambda]$ connaissant ses coordonnées cartésiennes (x, y, z) .

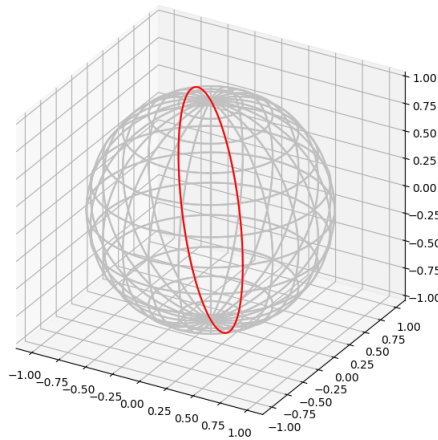
Question. Quelles sont les coordonnées sphériques $[r : \varphi : \lambda]$ du point $(x, y, z) = (1, 2, 3)$? Vérifie que si, à partir de ces $[r : \varphi : \lambda]$ et de la première question, tu calcules (x, y, z) tu retrouves bien $(1, 2, 3)$.

3. Tracer les méridiens et les parallèles.

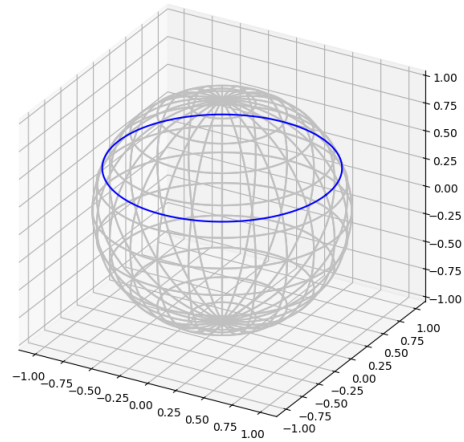
- (a) Programme une fonction `trace_meridien(r, lamb)` (ou mieux `trace_meridien(r, lamb, nbpoints=100, couleur='red')`) qui trace le cercle méridien, connaissant le rayon r et la longitude λ . (Figure de gauche ci-dessous.)

Indications.

- Définis N points $[r : \varphi : \lambda]$ où φ varie dans $[-\pi, \pi]$.
- Calcule les coordonnées (x, y, z) de chacun de ces points.
- Relie les points entre eux.

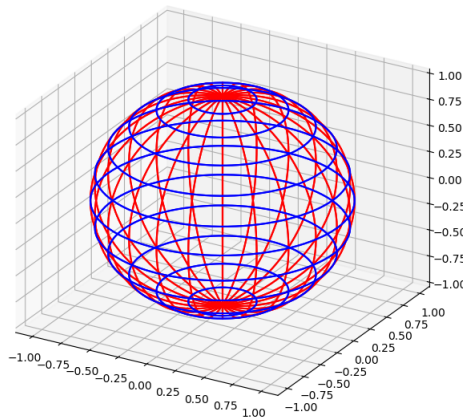


Un méridien.



Un parallèle.

- (b) Programme une fonction `trace_parallele(r, phi)` (ou mieux `trace_parallele(r, phi, nbpoints=100, couleur='blue')`) qui trace le cercle parallèle, connaissant le rayon r et la latitude φ . (Figure de droite ci-dessus.)
- (c) Programme une fonction `trace_meridiens_paralleles(r)` qui trace des méridiens et des parallèles sur la sphère de rayon r .



4. Grand cercle passant par deux points.

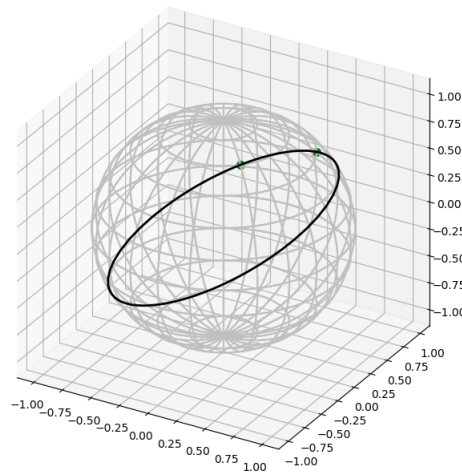
Problème. Quel est le trajet que doit parcourir un avion entre deux villes de la Terre ?

Mathématiquement, on se donne deux points P et Q sur la sphère de rayon r . On cherche le chemin le plus court tracé à la surface de la sphère qui va de P à Q . Réponse : c'est un des arcs du « grand cercle » passant par P et Q . Un *grand cercle* est un cercle de rayon r tracé sur la sphère ayant ce même rayon r (l'équateur et les méridiens sont des exemples de grands cercles).

Voici comment tracer ce grand cercle :

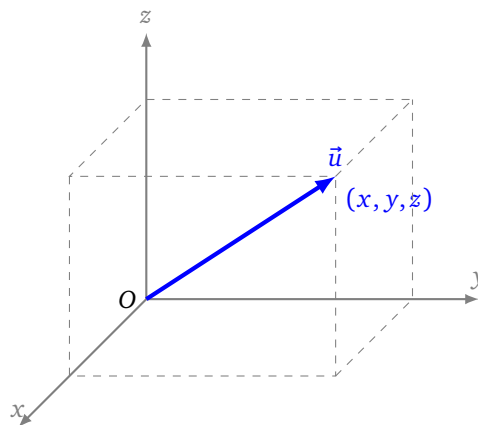
- On considère les points P et Q comme des vecteurs : $\vec{u} = \overrightarrow{OP}$ et $\vec{v} = \overrightarrow{OQ}$.
- On considère le vecteur tournant $\vec{w}(t) = \cos(t)\vec{u} + \sin(t)\vec{v}$, pour $t \in [0, 2\pi]$.
- On transforme le vecteur $\vec{w}(t)$ en un vecteur de norme r : $\vec{w}'(t) = r \frac{\vec{w}(t)}{\|\vec{w}(t)\|}$.
- Le point $R(t)$ à l'extrémité de $\vec{w}'(t)$ est sur le grand cercle passant par P et Q (autrement dit $R(t)$ est tel que $\vec{w}'(t) = \overrightarrow{OR(t)}$).

Programme une fonction `trace_grand_cercle(P,Q)` qui affiche le grand cercle passant par les deux points P et Q .



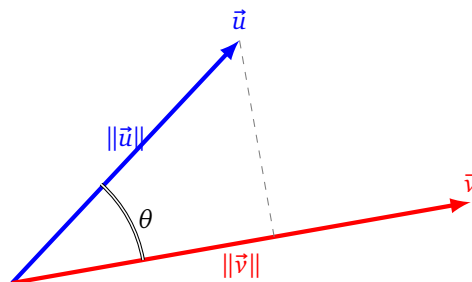
Cours 5 (Vecteurs).

On se place dans un repère orthonormé direct $(O, \vec{i}, \vec{j}, \vec{k})$. Un vecteur $\vec{u}$ est représenté par trois coordonnées (x, y, z) .



Soient $\vec{u} = (x, y, z)$, $\vec{v} = (x', y', z')$ deux vecteurs.

- **Addition.** Le vecteur $\vec{u} + \vec{v}$ a pour coordonnées $(x + x', y + y', z + z')$.
- **Multipliation par un scalaire.** Soit $k \in \mathbb{R}$. Alors $k\vec{u}$ a pour coordonnées (kx, ky, kz) .
- **Produit scalaire.** $\vec{u} \cdot \vec{v} = xx' + yy' + zz'$. C'est un nombre réel qui mesure la colinéarité des vecteurs $\vec{u}$ et $\vec{v}$.



- **Norme.** $\|\vec{u}\| = \sqrt{x^2 + y^2 + z^2}$. On a aussi $\|\vec{u}\| = \sqrt{\vec{u} \cdot \vec{u}}$. La norme est la longueur du vecteur $\vec{u}$.
- **Vecteurs orthogonaux.** Deux vecteurs $\vec{u}$ et $\vec{v}$ sont orthogonaux si et seulement si $\vec{u} \cdot \vec{v} = 0$.

- **Angle entre deux vecteurs.** La formule suivante permet de calculer l'angle θ entre deux vecteurs $\vec{u}$ et $\vec{v}$ non nuls :

$$\vec{u} \cdot \vec{v} = \|\vec{u}\| \|\vec{v}\| \cos(\theta)$$

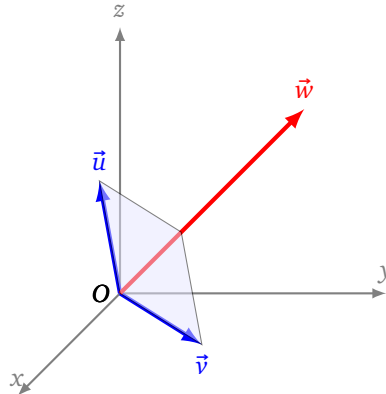
Ainsi

$$\theta = \arccos\left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \|\vec{v}\|}\right)$$

- **Produit vectoriel.** $\vec{u} \wedge \vec{v}$ est le vecteur de coordonnées

$$(yz' - y'z, zx' - z'x, xy' - x'y)$$

Ce vecteur est orthogonal au vecteur $\vec{u}$ et au vecteur $\vec{v}$. Autrement dit $\vec{w} = \vec{u} \wedge \vec{v}$ est orthogonal au plan qui contient $\vec{u}$ et $\vec{v}$.



- **Produit mixte.** C'est le nombre réel associé à trois vecteurs $\vec{u}$, $\vec{v}$ et $\vec{w}$, défini à l'aide d'un produit vectoriel puis d'un produit scalaire :

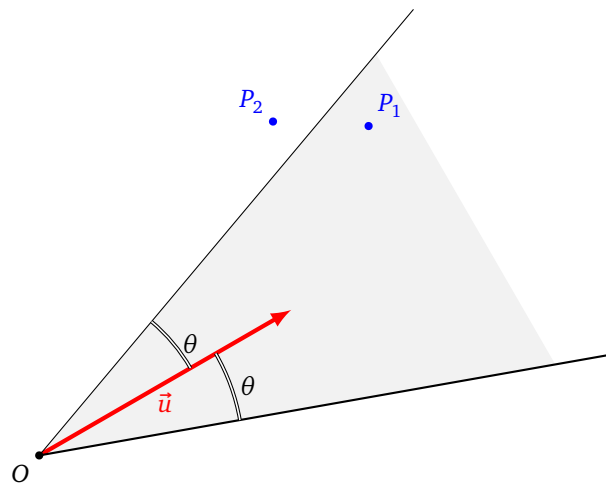
$$(\vec{u} \wedge \vec{v}) \cdot \vec{w}$$

Activité 4 (Vecteurs).

Objectifs : utiliser les vecteurs pour des calculs dans l'espace.

1. Norme et produit scalaire.

- Programme une fonction `produit_scalaire(u,v)` qui calcule le produit scalaire de deux vecteurs $\vec{u}$ et $\vec{v}$ de $\mathbb{R}^3$. La variable `u` contient un triplet (x, y, z) représentant les coordonnées de $\vec{u}$, de même pour `v`.
- Programme une fonction `norme(u)` qui calcule la norme d'un vecteur $\vec{u}$ de $\mathbb{R}^3$.
- Programme une fonction `angle(u,v)` qui calcule l'angle entre des vecteurs $\vec{u}$ et $\vec{v}$.
- Calcule la norme de $\vec{u} = (1, 2, 3)$ et de $\vec{v} = (1, 0, 1)$. Puis calcule le produit scalaire entre ces deux vecteurs, et enfin l'angle entre ces vecteurs (en radians et en degrés).
- Application : points visibles ou invisibles.* Un observateur regarde dans une direction, selon son champ de vision, quels sont les points visibles ?
 - *Modélisation.* L'observateur est au point $O = (0, 0, 0)$. Il regarde dans la direction $\vec{u}$. Son champ de vision est déterminé par un angle θ . Sur le schéma ci-dessous le point P_1 est visible par l'observateur mais pas le point P_2 . Bien sûr, dans l'espace, la zone visible est un cône (et pas un secteur comme sur le schéma).



- *Solution.* Un point P est visible depuis O si et seulement si l'angle entre $\vec{u}$ et $\vec{OP}$ est plus petit que θ , c'est-à-dire :

$$|\text{angle}(\vec{u}, \vec{OP})| \leq \theta$$

- *Question.* Un observateur a un angle de vision $\theta = 50^\circ$, il regarde dans la direction $(1, 1, 1)$. Parmi les points $P_1 = (1, 1, 2)$, $P_2 = (-1, -1, -2)$, $P_3 = (80, 10, 0)$, $P_4 = (85, 10, 0)$ lesquels sont visibles à ses yeux ?

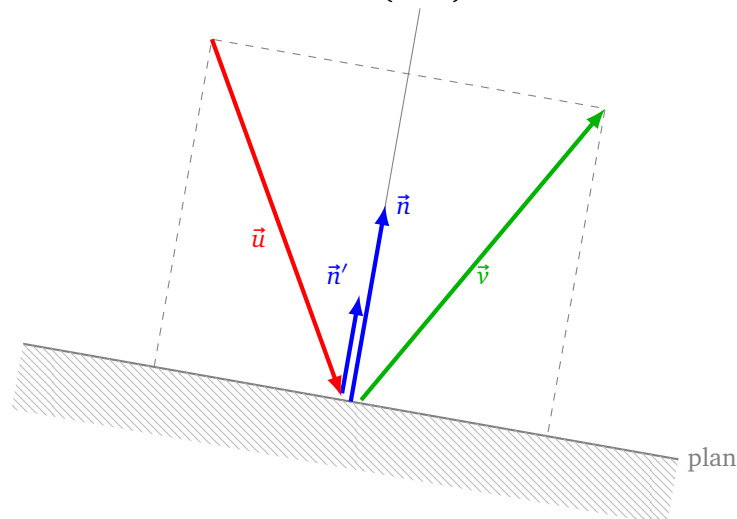
(f) *Application : rebond sur un plan.* Une balle arrive sur un plan et rebondit. Quelle est la nouvelle trajectoire de cette balle ?

- *Modélisation.* La balle arrive selon un vecteur vitesse $\vec{u}$, le plan est représenté par un vecteur normal $\vec{n}$.
- *Solution.* La balle repart selon le vecteur $\vec{v}$ qui est un symétrique de $\vec{u}$ par rapport à $\vec{n}$ et est donné par la formule suivante : on commence par transformer $\vec{n}$ en un vecteur de norme 1 :

$$\vec{n}' = \frac{\vec{n}}{\|\vec{n}\|}$$

puis on a :

$$\vec{n} = \vec{u} - 2(\vec{u} \cdot \vec{n}')\vec{n}'$$



- *Question.* Une balle arrive selon le vecteur $\vec{u} = (1, 2, -1)$ et rebondit sur un plan ayant pour vecteur normal $\vec{n} = (1, 1, 1)$. Selon quelle direction $\vec{v}$ repart-elle ?

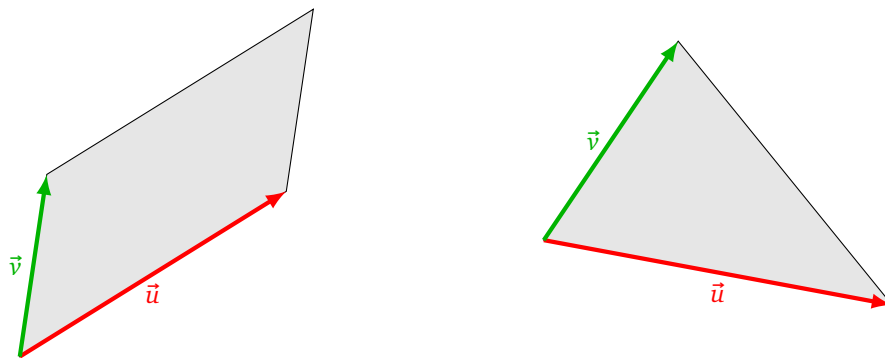
2. Produit vectoriel.

- (a) Programme une fonction `produit_vectoriel(u, v)` qui calcule le produit vectoriel entre deux vecteurs $\vec{u}$ et $\vec{v}$ de $\mathbb{R}^3$. Le résultat est un vecteur $\vec{w}$ renvoyé sous la forme d'un triplet de coordonnées (x, y, z) .
- (b) Calcule le produit vectoriel $\vec{w}$ de $\vec{u} = (1, 2, 3)$ et $\vec{v} = (1, 0, 1)$. Vérifie à l'aide du produit scalaire que $\vec{w}$ est orthogonal à $\vec{u}$ et à $\vec{v}$.
- (c) *Application : équation d'un plan.* On considère le plan P défini par les trois points $O = (0, 0, 0)$, $A = (-1, 2, 5)$, $B = (2, 0, 3)$. Calcule un vecteur $\vec{n} = (a, b, c)$ normal à ce plan P . Une équation du plan est alors $ax + by + cz = 0$.
- (d) *Application : surface d'un triangle et d'un parallélogramme de l'espace.* La surface d'un parallélogramme de l'espace déterminé par deux vecteurs $\vec{u}$ et $\vec{v}$ est

$$S_P = \|\vec{u} \wedge \vec{v}\|$$

La surface du triangle de l'espace déterminé par ces mêmes vecteurs est la moitié :

$$S_T = \frac{1}{2} \|\vec{u} \wedge \vec{v}\|$$



Calcule la surface du parallélogramme (puis du triangle) déterminé par les vecteurs $\vec{u} = (1, 2, -5)$ avec $\vec{v} = (1, -2, 4)$ (il est préférable de donner la réponse à l'aide de la racine carrée d'un entier plutôt qu'une valeur approchée).

3. Produit mixte.

- (a) Programme une fonction `produit_mixte(u, v, w)` qui calcule le produit mixte de trois vecteurs $\vec{u}$, $\vec{v}$, $\vec{w}$ de $\mathbb{R}^3$. On rappelle la formule du produit mixte

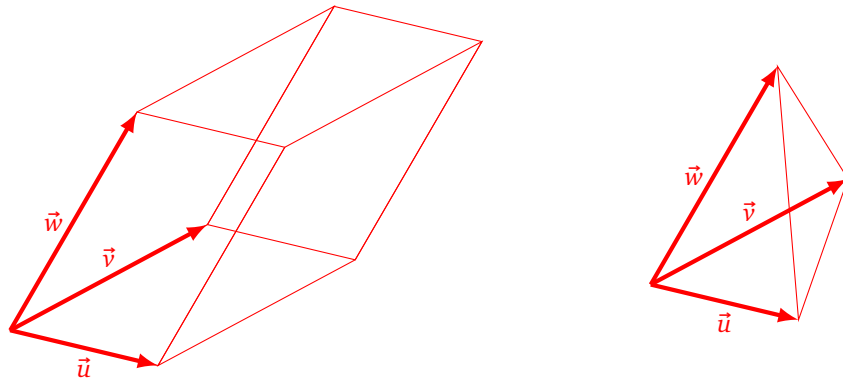
$$(\vec{u} \wedge \vec{v}) \cdot \vec{w}$$

- (b) Calcule le produit mixte de $\vec{u} = (1, 2, 3)$, $\vec{v} = (1, 0, 1)$ et $\vec{w} = (4, 1, 0)$.
- (c) *Application : volume.* Le volume d'un parallélépipède de l'espace déterminé par trois vecteurs $\vec{u}$, $\vec{v}$ et $\vec{w}$ est

$$V_P = |\text{produit_mixte}(\vec{u}, \vec{v}, \vec{w})|$$

Le volume du tétraèdre déterminé par ces mêmes vecteurs est $1/6$ du volume précédent :

$$V_T = \frac{1}{6} |\text{produit_mixte}(\vec{u}, \vec{v}, \vec{w})|$$

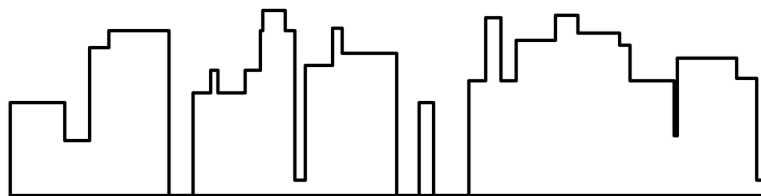


Calcule le volume du parallélépipède (puis du tétraèdre) déterminé par les vecteurs $\vec{u} = (1, 0, 0)$, $\vec{v} = (1, 1, 0)$ et $\vec{w} = (1, 1, 1)$.

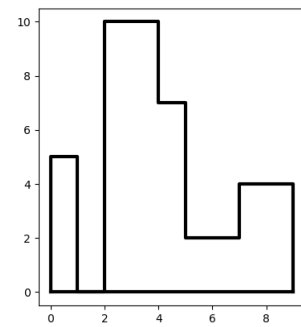
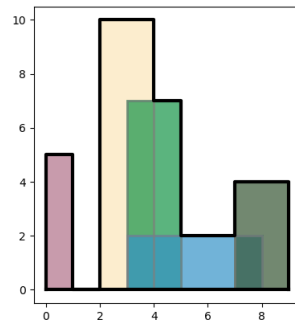
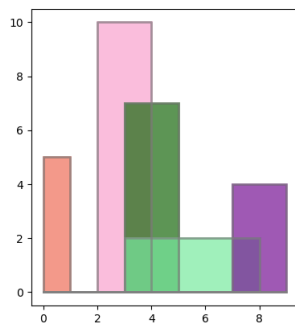
Activité 5 (Skyline).

Objectifs : tracer la skyline d'une ville, c'est-à-dire le contour apparent de ses gratte-ciels.

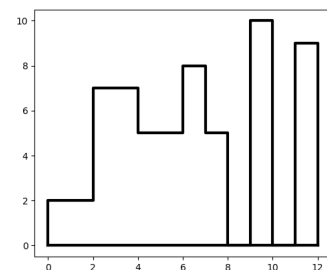
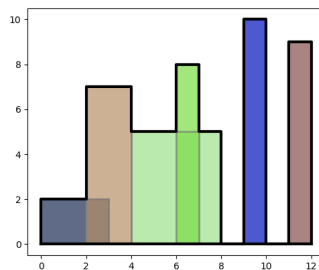
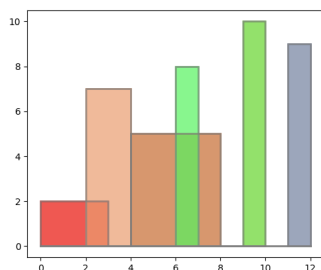
Le problème est simple : on souhaite dessiner le contour apparent d'une ville constituée de gratte-ciels.



Voici à gauche les immeubles, au centre on dessine le contour, à droite on garde uniquement ce contour.



Voici un autre exemple.



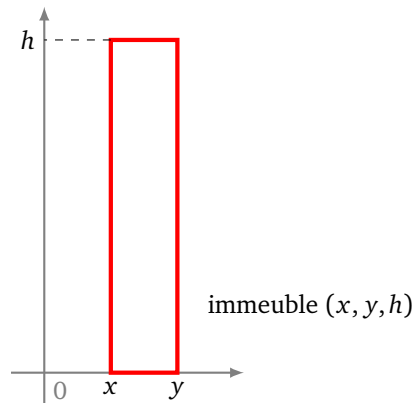
Ce qui serait formidable c'est de ne pas lire la suite de l'activité et que tu te débrouilles tout seul pour modéliser et résoudre ce problème !

Indications.

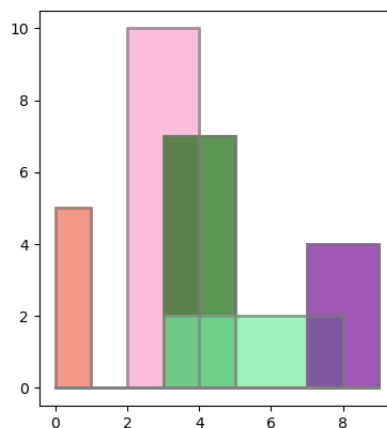
- Il faut d'abord comprendre que ce problème, qui devrait être un problème en trois dimensions, se ramène à un problème à deux dimensions seulement : qu'un immeuble soit devant ou derrière un autre ne change pas le contour.
- Il faut aussi avoir conscience que même si le problème est facile à énoncer, trouver une solution n'est pas simple !
- C'est plus important que tu trouves un algorithme seul, même si il n'est pas parfait, plutôt que de suivre la solution proposée ci-dessous qui est assez optimale.

Modélisation.

- On modélise un gratte-ciel par un triplet (x, y, h) où x est l'abscisse du côté gauche de l'immeuble, y est l'abscisse du côté droit, h est sa hauteur.



- Une ville est donc une liste d'immeubles. Voici l'exemple pour cette activité :
`immeubles = [(0,1,5), (2,4,10), (3,5,7), (3,8,2), (7,9,4)]`

**Principe.** L'idée mise en œuvre ci-dessous est la suivante.

- Tout d'abord on récupère tous les bords : ce sont les x ou y correspondant à un côté d'immeuble.
- Pour chacun de ces bords, on associe la liste des immeubles qui commencent ou finissent ici. On utilisera un dictionnaire.
- Pour calculer la *skyline*, on parcourt les bords de la gauche vers la droite. On calcule les listes de tous les immeubles actifs : ce sont les immeubles présents à cette abscisse.
- Chaque bord détermine donc la hauteur maximale actuelle. Si cette hauteur diffère de celle du bord précédent, on a un point de la *skyline* !

Voici le travail décomposé en étapes.

1. Programme une fonction `hauteur_max_immeubles(immeubles)` qui renvoie la hauteur maximale d'une liste d'immeubles (ou 0 si la liste est vide).

Par exemple avec :

```
immeubles = [(0,1,5), (2,4,10), (3,5,7), (3,8,2), (7,9,4)]
```

la fonction renvoie la hauteur maximale $h = 10$.

2. Programme une fonction `calcul_bords(immeubles)` qui renvoie la liste de tous les bords : dans l'ordre croissant et sans redondance. Un *bord* est une abscisse x ou y d'un immeuble.

Par exemple avec :

```
immeubles = [(0,1,5), (2,4,10), (3,5,7), (3,8,2), (7,9,4)]
```

la fonction renvoie la liste :

```
[0, 1, 2, 3, 4, 5, 7, 8, 9]
```

3. Programme une fonction `dictionnaire_bords_immeubles(immeubles)` qui renvoie un dictionnaire associant à chaque abscisse la liste des numéros d'immeubles ayant un bord ici. Une clé est donc un bord, la valeur une liste de numéros d'immeubles.

Par exemple avec :

```
immeubles = [(0,1,5), (2,4,10), (3,5,7), (3,8,2), (7,9,4)]
```

la fonction renvoie le dictionnaire :

```
dico = {0: [0], 1: [0], 2: [1], 4: [1],
        3: [2, 3], 5: [2], 8: [3], 7: [4], 9: [4]}
```

Par exemple `dico[2]` vaut `[1]` cela signifie que seul l'immeuble numéro 1 a un bord à l'abscisse 2. Autre exemple `dico[3]` vaut `[2, 3]`, cela veut dire que les immeubles numéros 2 et 3 ont un bord à l'abscisse 3.

Indications.

- La manipulation d'un dictionnaire est expliquée dans la fiche « Le mot le plus long ».
 - Pars d'un dictionnaire vide `dico = {}`.
 - Pour chaque immeuble récupère les valeurs x et y :
 - si x n'est pas déjà une clé (test « `x not in dico` ») crée une nouvelle liste contenant le numéro de l'immeuble (`dico[x] = [i]`),
 - si x est déjà une clé (test « `x in dico` ») ajoute le numéro de l'immeuble à la liste (`dico[x].append(i)`),
 - fais la même chose avec y .
4. Programme une fonction `calcul_skyline(immeubles)` qui renvoie la liste des points formant le contour. Pour l'exemple des questions précédentes le contour est :

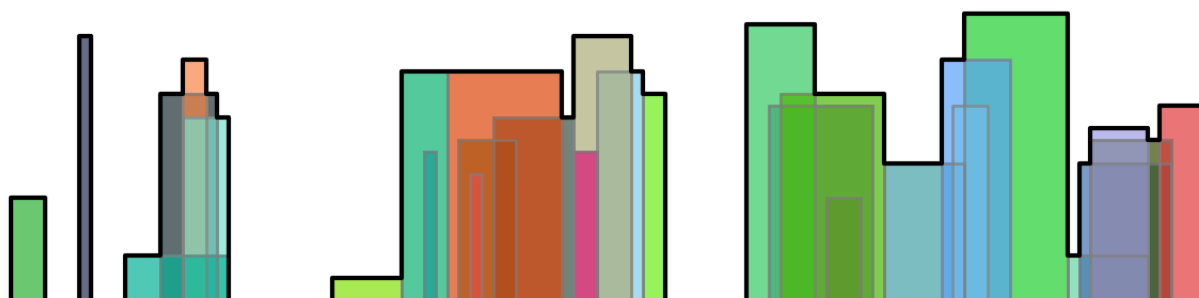

```
[(0, 0), (0, 5), (1, 5), (1, 0), (2, 0), (2, 10), (4, 10),
 (4, 7), (5, 7), (5, 2), (7, 2), (7, 4), (9, 4), (9, 0)]
```

Voici l'algorithme :

- Calculer la liste des bords et le dictionnaire bords/immeubles.
- Initialiser une liste vide pour la *skyline* et une pour les immeubles actifs.
- Initialiser une hauteur `h_avant` à 0.
- Pour chaque x dans la liste des bords :
 - calculer la liste des immeubles actifs (pour chaque i dans `dico[x]` on ajoute ou on retire l'immeuble numéro i s'il est absent ou déjà présent),
 - calculer la nouvelle hauteur maximale `h_apres` des immeubles actifs,
 - si `h_avant` et `h_apres` diffèrent alors ajouter à la *skyline* les deux points (x, h_{avant}) et (x, h_{apres}) ,
 - `h_avant` $\leftarrow$ `h_apres`

5. Programme enfin l'affichage des immeubles et de la *skyline* !

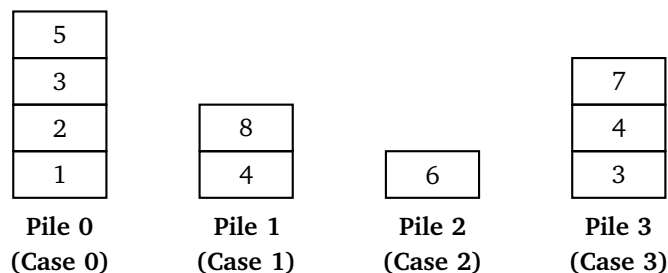
Voici un exemple généré au hasard.



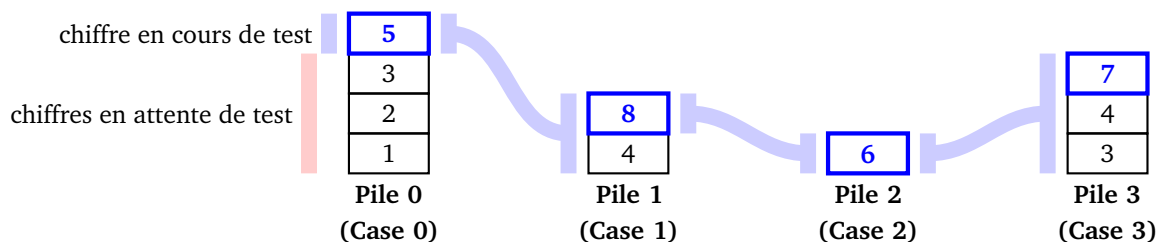
Tu vas programmer un algorithme qui complète entièrement une grille de sudoku. La méthode utilisée est la recherche par l'algorithme du « retour en arrière ».

Cours 1 (Recherche par retour en arrière).

On cherche la solution à une grille de sudoku (les règles et les détails sont donnés dans l'activité 3). Chaque case correspond à une pile, chaque pile correspond à une liste de chiffres possibles.



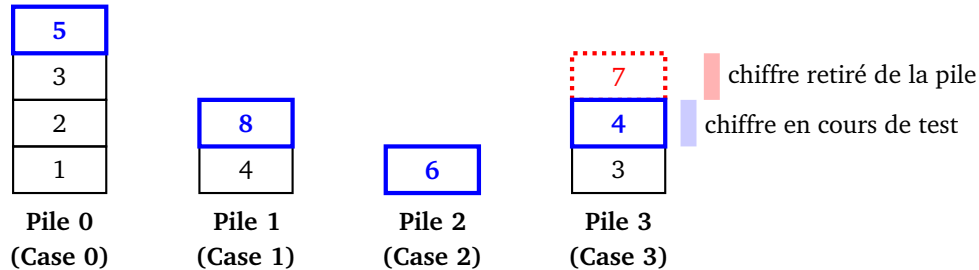
- On liste les chiffres possibles pour la case numéro 0 (par exemple la figure indique que les chiffres 1, 2, 3 ou 5 sont possibles).
- Parmi ces chiffres, on teste la configuration où le chiffre est le dernier de la liste (donc ici on suppose que dans la case numéro 0 on place le chiffre 5).
- Suivant cette hypothèse, on liste les choix possibles pour la case numéro 1 (ici 4 et 8 sont possibles). Attention, chaque pile dépend des piles précédentes. Ici pour la pile 0 formée de [1, 2, 3, 5], avec 5 en haut, la pile 1 est [4, 8].)
- On choisit pour cette pile numéro 1 de tester le dernier chiffre de la liste [4, 8] (c'est comme si on plaçait 8 dans la case numéro 1).
- On continue. Pour les possibilités des cases suivantes, on tient compte de la configuration en cours de test (donc ici pour la case suivante, la figure ci-dessous avec les flèches indique que seul le chiffre 6 est possible, et cela tient compte du 5 en case 0 et du 8 en case 1).
- Si on continue et on arrive à remplir toute la grille, c'est gagné !



- Si à un moment on est bloqué, c'est-à-dire qu'il n'y a aucun chiffre possible pour la case suivante, alors on effectue un retour en arrière. Voyons un exemple. Partons de la configuration en cours de test

sur la figure : on a placé les chiffres 5, 8, 6 et 7 dans les quatre premières cases. Ce sont les chiffres en haut des piles.

Imaginons que pour la case suivante il n'y ait aucune possibilité. Alors on raye le chiffre 7 de la liste des possibilités pour la case numéro 3, c'est-à-dire qu'on le retire de la pile numéro 3. On suppose maintenant que le chiffre en cours de test pour la case numéro 3 est le chiffre 4. On repart en avant en cherchant les chiffres possibles pour la case suivante...



Cours 2 (Quatre problèmes simples).

Les problèmes suivants n'ont pas d'intérêt en soi, mais ils sont beaucoup plus simples et vont nous permettre de tester notre algorithme de recherche par retour en arrière.

- **Problème 1.** Trouver toutes les listes de quatre chiffres telles que :

- le chiffre au rang 0 est 1, 2, 3 ou 4 ;
- le chiffre au rang 1 est 5 ou 6 ;
- le chiffre au rang 2 est 7 ou 8 ;
- le chiffre au rang 3 est 9.

Exemple : [3, 6, 7, 9] est une solution.

- **Problème 2.** Trouver toutes les listes de quatre chiffres telles que :

- le chiffre au rang 0 est 1, 2 ou 3 ;
- le chiffre au rang 1 est le double du chiffre au rang 0 ;
- le chiffre au rang 2 est 5, 7 ou 9 ;
- le chiffre au rang 3 est le même que le chiffre au rang 2.

Exemple : [2, 4, 7, 7] est une solution.

- **Problème 3.** Trouver toutes les listes de quatre chiffres telles que :

- le chiffre au rang 0 est 1, 3, 5, 7 ou 9 ;
- le chiffre au rang 1 est 2 ou 4 si le chiffre au rang 0 est supérieur ou égal à 5, et 6 ou 8 sinon ;
- le chiffre au rang 2 est la moitié du chiffre au rang 1 ;
- le chiffre au rang 3 est le chiffre au rang 0 diminué de 1 ou bien est 9.

Exemple : [5, 2, 1, 4] est une solution.

- **Problème 4.** Trouver toutes les listes de quatre chiffres telles que chacun des chiffres soit 0 ou 1 (sans autres contraintes). Exemple : [0, 1, 0, 0] est une solution.

Activité 1 (Recherche par retour en arrière).

Objectifs : programmer la recherche de solutions par la méthode du retour en arrière et l'appliquer ici à des problèmes simples.

On a besoin de deux variables globales :

- le nombre maximum de piles n (initialisé par exemple par $n = 4$),
- la liste des piles `les_piles`.

Par exemple la configuration des piles :

4			
1	5		8
3	1	2	7
Pile 0	Pile 1	Pile 2	Pile 3

correspond à :

```
les_piles = [ [3,1,4], [1,5], [2], [7,8] ]
```

1. **Haut des piles.** Programme une fonction `haut_des_piles()` qui renvoie la liste des éléments en haut de chaque pile. Par exemple si `les_piles = [ [3,1,4], [1,5], [2], [7,8] ]` alors `haut_des_piles()` renvoie `[4, 5, 2, 8]`.
2. **Choix.** Programme une fonction `choix(i)` (qu'il faudra changer à chaque problème) et qui renvoie une liste afin d'initialiser la pile du rang i , c'est-à-dire la liste des choix possibles. Dans cette question, on répond au problème numéro 1, c'est-à-dire que `choix(i)` renvoie :

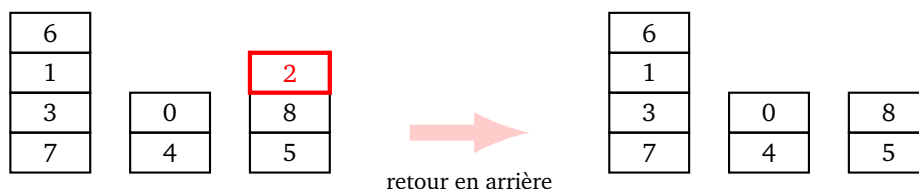
- `[1,2,3,4]` si $i = 0$,
- `[5,6]` si $i = 1$,
- `[7,8]` si $i = 2$,
- `[9]` si $i = 3$.

Exemple. En partant d'une liste de piles vide `les_piles = []`, utilise cette fonction, pour arriver à la configuration où `les_piles` vaut `[ [1,2,3,4], [5,6], [7,8], [9] ]`.

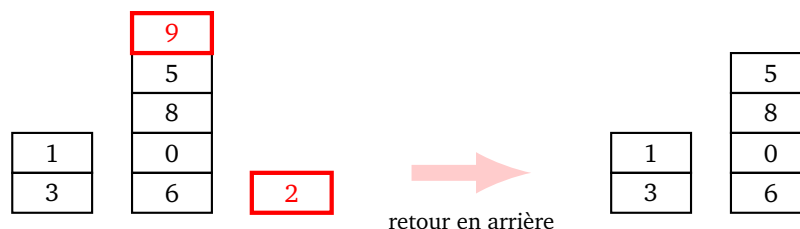
3. **Retour en arrière.** Programme une fonction `retour()` qui exécute un retour en arrière dans notre liste des piles `les_piles`.

On a besoin de faire un retour en arrière soit lorsque l'on se trouve dans une configuration bloquante, soit lorsque l'on a trouvé une solution et que l'on repart à la recherche d'une autre solution.

- La situation la plus simple, c'est lorsqu'il reste au moins deux éléments sur la dernière pile : il suffit de retirer l'élément en haut de la dernière pile.



- Par contre s'il n'y a qu'un seul élément sur la dernière pile, il faut supprimer cette pile et retirer un élément à la pile précédente.



- Mais il se peut que sur plusieurs piles de la fin, il n'y ait qu'un seul élément. Il faut alors supprimer ces piles et retirer un élément sur une pile qui en contient deux.



Pour tenir compte de tous ces cas, l'algorithme est le suivant. Il fonctionne en deux étapes : tout d'abord on élimine (si besoin) toutes les piles de la fin qui sont seulement de taille 1 ; puis on retire l'élément du haut d'une pile de taille au moins 2.

Algorithme.

- Action : effectue un retour en arrière sur les piles, ce qui modifie la variable globale `les_piles`.
- On note r le rang de la dernière pile.
- Tant que $r \geq 0$ et que la taille de la pile numéro r vaut 1 :
 - supprimer la dernière pile,
 - faire $r \leftarrow r - 1$
- Ensuite, si $r \geq 0$, supprimer l'élément du haut de la pile numéro r .

4. Recherche des solutions.

On recherche une configuration de n piles, qui va donner une solution en prenant chaque élément en haut des piles. Comment constituer la liste des piles ? La première pile est constituée des choix possibles pour le rang 0 à l'aide de la commande `choix(0)`. Tant que le nombre maximum de piles n'est pas atteint on essaie de rajouter une nouvelle pile (toujours avec la fonction `choix()`) si c'est possible on le fait, sinon on effectue un retour en arrière. Si on arrive au nombre maximal de pile, on a une solution. Si on arrive à une liste vide de piles (après des retours en arrière), c'est qu'on a testé toutes les possibilités.

Algorithme.

- — Action : recherche une, ou toutes les solutions, au problème déterminé par la fonction `choix()`. Une solution est formée de n éléments.
- — Sortie : affichage des solutions.
- On va utiliser la variable globale `les_piles`, initialisée par la liste vide.
- Ajouter dans `les_piles` une première pile donnée par `choix(0)`.
- On définit un drapeau `termine` à la valeur « Faux ».
- Tant que `termine` ne vaut pas « Vrai » :
 - Noter r la longueur de la liste des piles `les_piles`.
 - Si $r = 0$, alors mettre `termine` à la valeur « Vrai » (les piles sont toutes vides, il n'y a plus rien à tester).
 - Si $0 < r < n$ (le nombre maximum de piles n'est pas atteint) :
 - On calcule la prochaine pile `nouv_pile` par `choix(r)`.
 - Si `nouv_pile` n'est pas vide (c'est qu'il y a des possibilités), alors ajouter `nouv_pile` à la liste de toutes les piles `les_piles`,
 - sinon (il n'y a aucune possibilité, on est bloqué), effectuer un retour en arrière par la commande `retour()`.
 - Si $r = n$ (le nombre maximum de piles est atteint) alors on obtient une solution à notre problème en prenant chaque élément en haut des piles (par la fonction `haut_des_piles()`). On affiche cette solution. Puis on fait l'une des deux actions suivantes (commenter celle qui ne sert pas) :
 - si on cherche une seule solution, alors mettre `termine` à « Vrai » (on stoppe la recherche),
 - si on veut toutes les solutions, alors exécuter un retour en arrière par la commande `retour()`.

5. D'autres problèmes.

Modifie la fonction `choix()` afin de répondre aux problèmes donnés dans le cours auparavant.

(a) **Problème numéro 2.**

- Il faut commencer par récupérer la liste des éléments en haut des piles renvoyée par la commande `haut = haut_des_piles()`.
- `choix(0)` donne la liste `[1, 2, 3]`,
- `choix(1)` donne la liste composée d'un seul élément : `[2*haut[0]]` (cela dépend donc de l'état des piles),
- `choix(2)` donne la liste `[5, 7, 9]`,
- `choix(3)` donne la liste composée d'un seul élément : `[haut[2]]` (cela dépend donc de l'état des piles).

(b) **Problème numéro 3.**

- Il faut commencer par récupérer la liste `haut` par la commande `haut_des_piles()`.
- `choix(0)` donne la liste `[1, 3, 5, 7, 9]`,
- `choix(1)` donne `[2, 4]` si `haut[0] ≥ 5`, ou `[6, 8]` sinon,
- `choix(2)` donne la liste composée du seul élément `haut[1] // 2`,
- `choix(3)` donne la liste composée des deux éléments `haut[0] - 1` et `9`.

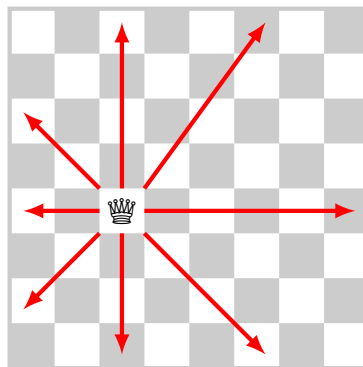
(c) **Problème numéro 4.**

Comment fais-tu pour afficher tous les nombres binaires sous la forme de listes de 4 *bits* ? Tu dois afficher toutes les listes possibles : `[0, 0, 0, 0]`, `[0, 0, 0, 1]`,... jusqu'à `[1, 1, 1, 1]`.

Cours 3 (Le problème des huit reines).

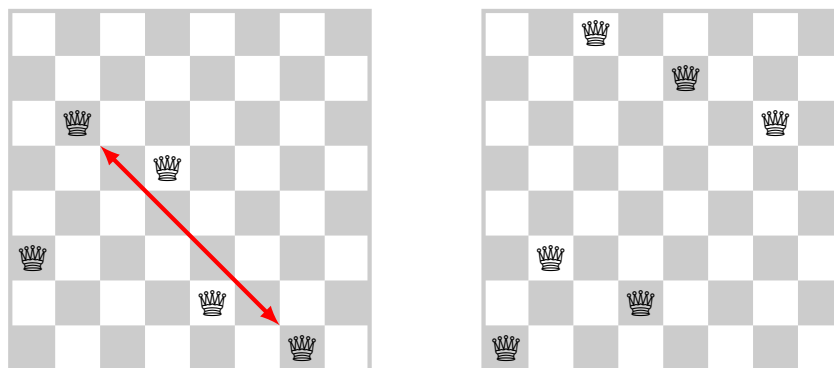
Sur un échiquier de taille 8×8 , une reine peut se positionner sur n'importe quelle case (noire ou blanche). Ensuite elle peut se déplacer pour capturer une pièce :

- sur n'importe quelle case de la même ligne,
- sur n'importe quelle case de la même colonne,
- sur n'importe quelle case d'une des deux diagonales.



Problème des huit reines. Placer 8 reines sur un échiquier de sorte qu'aucune reine ne puisse en capturer une autre.

Exemples. Voici à gauche un exemple d'une configuration qui ne conviendra pas. En effet, il y a deux reines situées sur une même diagonale. L'une peut donc capturer l'autre. À droite, un exemple de configuration avec 6 reines, aucune ne pouvant en capturer une autre. Comme il y a seulement 6 reines, et qu'il n'est pas possible d'en rajouter une autre, cela ne répond pas au problème.



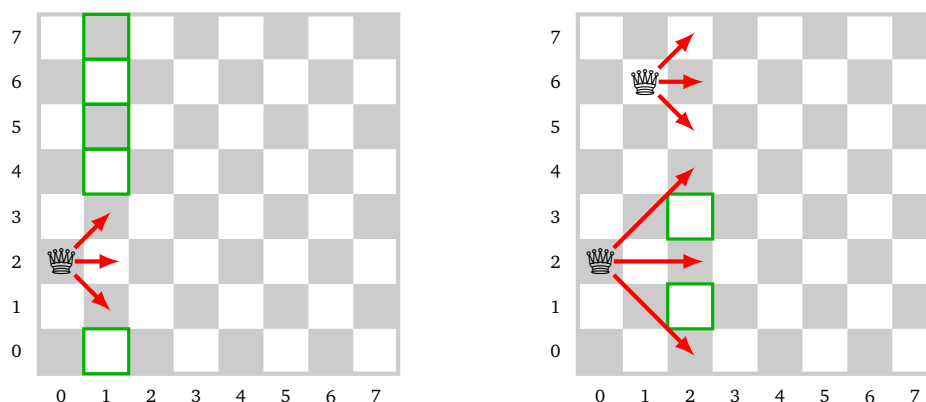
Méthode de recherche. Si on veut tester toutes les possibilités sans réfléchir, alors il y a 64 possibilités pour placer la première reine, 63 pour la seconde... soit un nombre total de configurations qui vaut :

$$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178\,462\,987\,637\,760$$

C'est beaucoup trop, même pour un ordinateur !

Pour diminuer le nombre de configurations à tester, nous allons tenir compte des reines en place, avant d'en ajouter une nouvelle. Pour cela on va placer une reine dans la colonne numéro 0, puis une reine dans la colonne numéro 1... (il ne peut y avoir qu'une seule reine par colonne).

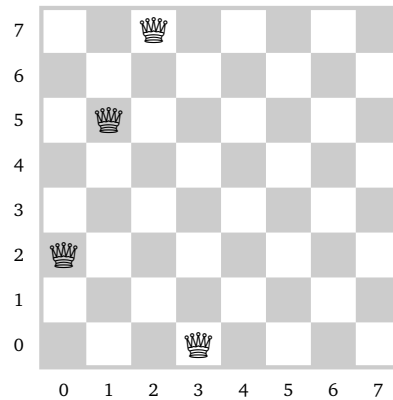
L'idée est donc de tester une position pour la colonne numéro 0 (par exemple en ligne 2, voir les figures ci-dessous). On cherche ensuite quelles sont les positions possibles pour placer une reine sur la colonne numéro 1 : il y a 5 cases possibles (figure de gauche). Par exemple on choisit de placer une reine en colonne numéro 1, ligne 6 (figure de droite). Alors pour la colonne numéro 2, en tenant compte des deux reines déjà en place, il ne reste que 2 cases possibles. On retient que si on a déjà placé des reines, il reste un nombre assez limité de choix pour placer une reine dans la colonne suivante.



Modélisation.

On note n la taille de l'échiquier, qui est aussi le nombre de reines à placer. Les colonnes sont numérotées de $i = 0$ à $i = n - 1$, les lignes aussi. On va placer une reine dans chaque colonne en partant de la gauche. Par exemple la configuration (2, 5, 7, 0) signifie que l'on a placé :

- sur la colonne 0, une reine en ligne 2,
- sur la colonne 1, une reine en ligne 5,
- sur la colonne 2, une reine en ligne 7,
- sur la colonne 3, une reine en ligne 0.



Les configurations que l'on va tester sont toutes valides (aucune reine ne peut capturer une autre reine) mais pas forcément complètes (par exemple ici, il manque encore 4 reines).

Activité 2 (Le problème des huit reines).

Objectifs : écrire la fonction `choix()` pour résoudre le problème des huit reines par la méthode du retour en arrière.

- **Modélisation.**

- Définis $n = 8$.
- `les_piles` est la liste des piles. La pile numéro i contient une liste des lignes jouables pour la colonne numéro i .
- Pour une configuration donnée, on initialise la pile suivante par la commande `choix(i)`, qui correspond à la colonne numéro i .

- **Fonction `choix()`.** Modifie la fonction `choix()` de l'activité précédente de sorte que `choix(i)` renvoie la liste des cases jouables sur la colonne numéro i en fonction de la configuration en cours de test (donnée par le haut des piles).

- **Indications.**

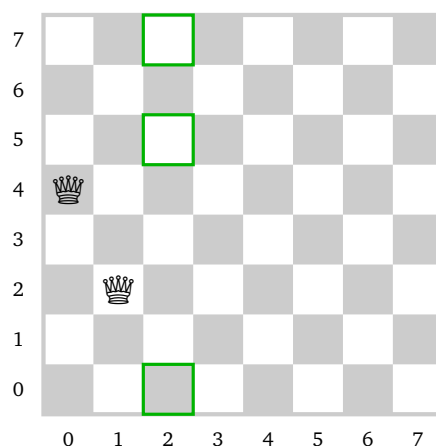
- Récupère la configuration en cours de test par la commande `haut = haut_des_piles()`.
- Quelles sont les lignes à éviter pour la prochaine colonne ?
 - Tu vas inclure dans une liste `eviter` la liste des positions interdites pour cette colonne.
 - Il faut éviter les lignes déjà occupées par une reine (directement données par `haut`, tu peux donc initialiser `eviter` à `haut`).
 - Il faut éviter d'être sur une diagonale d'une autre reine : par exemple si on a une reine en colonne j , et donc en ligne `haut[j]`, alors, sur la nouvelle colonne i , les cases en ligne `haut[j]+i-j` et `haut[j]-i+j` sont interdites (si bien sûr, ces nombres sont compris entre 0 et $n-1$).
 - Les lignes possibles sont alors le complément des lignes à éviter. Ainsi `choix(i)` renvoie la liste des entiers de 0 à $n-1$ qui n'apparaissent pas dans la liste `eviter`.
- Tu peux commencer par tester ton algorithme avec $n = 4$ (4 reines à placer sur un échiquier 4×4).

- **Exemples.**

- `choix(0)` doit renvoyer `[0, 1, 2, ..., 7]` car il n'y a aucune contrainte.
- Si par exemple :

```
les_piles = [ [0, 1, 2, 3, 4], [1, 2] ]
```

Que doit renvoyer `choix(2)` ? La configuration en cours de test est donnée par le haut des piles, donc sur la colonne 0 on a placé une reine en ligne 4, et sur la colonne 1 une reine en ligne 2. La configuration testée est donc la suivante :



Alors la commande `choix(2)` doit renvoyer la liste `[0, 5, 7]` qui sont les positions acceptables pour la colonne 2.

• **Questions.**

- Combien y a-t-il de solutions différentes au problème des 8 reines ?
- Combien y a-t-il de façons de placer 10 reines sur un échiquier 10×10 ?

Cours 4 (Sudoku).

Grille de sudoku. Une grille de sudoku est une grille de taille 9×9 qu'il faut remplir des chiffres de 1 à 9 en respectant les règles suivantes :

- sur chaque ligne, chaque chiffre n'apparaît qu'une seule fois,
- sur chaque colonne, chaque chiffre n'apparaît qu'une seule fois,
- dans chaque sous-bloc 3×3 , chaque chiffre n'apparaît qu'une seule fois.

Voici un exemple de grille à compléter (à gauche) et sa solution (à droite) :

	4		5		1		9	
7			4		3			6
	3			6			8	
		1				6		
5	2						1	9
		4				8		
	9			3			7	
4			7		2			8
	1		9		8		6	

Grille de départ

2	4	6	5	8	1	3	9	7
7	5	8	4	9	3	1	2	6
1	3	9	2	6	7	4	8	5
9	8	1	3	7	5	6	4	2
5	2	3	8	4	6	7	1	9
6	7	4	1	2	9	8	5	3
8	9	2	6	3	4	5	7	1
4	6	5	7	1	2	9	3	8
3	1	7	9	5	8	2	6	4

Grille résolue

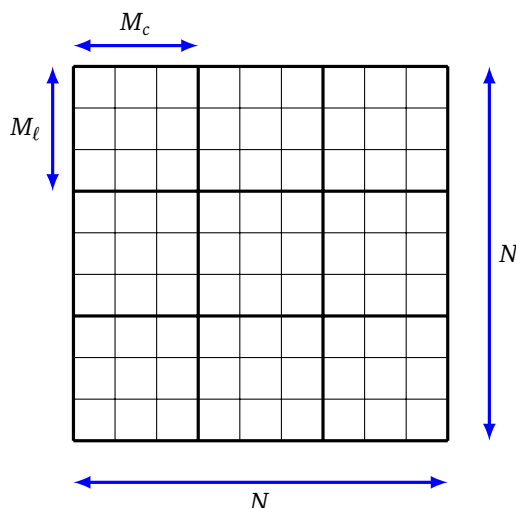
Unicité. Une grille partielle de sudoku doit être conçue de sorte que la solution soit unique.

Modélisation. On va considérer des grilles $N \times N$, contenant donc $n = N^2$ cases, à remplir par des chiffres de 1 à N . La grille est découpée en sous-blocs de taille $M_\ell \times M_c$ (contenant chacun exactement N cases). On s'intéresse principalement aux grilles avec $N = 9$ et $M_\ell = M_c = 3$. Mais pour les tests on peut commencer par des grilles 4×4 ($N = 4$ et $M_\ell = M_c = 2$) ou des grilles 6×6 ($N = 6$ et $M_\ell = 2$, $M_c = 3$).

1			
		2	
	3		4

Grille 4×4

		5		6	
6					
4			3		
	3				2
					1
	5		2		

Grille 6×6 

Numérotations des cases. Nous allons numéroter les cases de deux façons.

- *Coordonnées.* On peut identifier une case par ses coordonnées (i, j) , i étant le numéro de ligne et j le numéro de colonne (les rangs commencent à 0 et finissent à $N - 1$).
- *Ordre.* On numérote aussi les cases de $k = 0$ à $k = N^2 - 1$ en partant d'en haut à gauche.

	j								
	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									
5									
6									
7									
8									

Numérotation (i, j)

0	1	2	3	4	5	6	7	8
9	10	11	12					
18								
						61	62	63

Numérotation k

- *Conversion.* Voici les formules qui permettent de passer de la numérotation par les coordonnées (i, j) à la numérotation par l'ordre k :

$$k = i \cdot N + j \quad \text{et} \quad \begin{cases} i = k // N \\ j = k \% N \end{cases}$$

Modélisation. On modélise une grille de sudoku par un tableau $N \times N$. On associe à une case vide le chiffre 0. Par exemple le tableau de gauche encode la grille de droite.

1	0	0	0
0	0	2	0
0	0	0	0
0	3	0	4

1			
		2	
	3		4

Piles. On va trouver la solution à une grille de sudoku par une recherche avec retour en arrière. Pour cela nous utiliserons une suite de piles, avec une pile par case (donc 64 pour une grille classique). Dans la pile numéro 0 on liste les chiffres possibles pour la case numéro 0, dans la pile numéro 1 on liste les chiffres possibles pour la case numéro 1, en tenant compte du chiffre en haut de la pile numéro 0, qui

correspond à la configuration en cours de test, dans la pile numéro 2 on liste les chiffres possibles pour la case numéro 2...

Pour une case remplie dans la grille de départ, la pile associée ne contient que le seul chiffre de cette case.

Exemple. Voyons le démarrage de la recherche d'une solution avec une grille 2×2 .

1			
		2	
	3		4

Grille de départ

- La case numéro 0 contient un chiffre dans la grille de départ, donc la pile numéro 0 est réduite à la pile [1]. La liste des piles est pour l'instant réduite à une seule à pile : [[1]].
- La commande `choix(1)` nous donne les possibilités pour la case numéro 1. Les seuls chiffres possibles sont 2 et 4, donc la pile numéro 1 est [2, 4] et la liste des piles est maintenant [[1], [2, 4]].
- La configuration en cours de test est donnée par le haut des piles, donc pour l'instant c'est comme si la case numéro 1 contenait le chiffre 4 (grille de gauche ci-dessous).

1	4		

Haut des piles

1	4		
		2	
	3		4

Grille en cours de test

- La commande `choix(2)` tient compte aussi des chiffres de départ, la configuration en cours de test est donc donnée par la grille de droite ci-dessus. La seule possibilité pour la case numéro 2 est le chiffre 3, la liste des piles est maintenant [[1], [2, 4], [3]].
- La configuration de la grille en cours de test est donc la grille ci-dessous. Pour la case suivante (la numéro 3) il n'y a aucune possibilité (`choix(3)` renvoie une liste vide).

1	4	3	
		2	
	3		4

Grille en cours de test

aucune possibilité

- Comme on est bloqué, pour continuer notre recherche on effectue un retour en arrière : ce qui nous ramène à la liste des piles valant : [[1], [2]], c'est-à-dire à la configuration ci-dessous. Autrement dit, on a exclu la possibilité du chiffre 4 dans la case numéro 1.

1	2		
		2	
	3		4

Grille après retour en arrière

- On liste alors les possibilités pour la case suivante (chiffre 3 ou 4 dans la case numéro 2), etc.

Activité 3 (Sudoku).

Objectifs : programmer la résolution automatique d'une grille de sudoku.

1. Grille de départ.

- Définis des variables globales N , $M1$, Mc .
- Initialise ce qui va être la grille de départ par :

$$\text{grille_depart} = [[0 \text{ for } j \text{ in range}(N)] \text{ for } i \text{ in range}(N)]$$
- Puis par des commandes du type $\text{grille_depart}[i][j] = \text{valeur}$, remplis la grille de départ.
- Initialise toutes ces variables pour définir la grille de départ de l'exemple :

$$[[1, 0, 0, 0], [0, 0, 2, 0], [0, 0, 0, 0], [0, 3, 0, 4]]$$

1			
		2	
	3		4

- 2. Affichage.** Programme une fonction `voir_grille(grille)` qui affiche à l'écran une grille (éventuellement incomplète) de sudoku. Par exemple la grille de la question précédente peut s'afficher ainsi :

```

1 _ _ _
_ _ 2 _
_ _ _ _
_ 3 _ 4

```

- 3. Conversions.** Programme deux fonctions `case_vers_numero(i, j)` et `numero_vers_case(k)` qui effectuent les conversions entre la numérotation par les coordonnées (i, j) et la numérotation par ordre k (voir les formules données dans le cours).
- 4. Liste vers grille.** Programme une fonction `liste_vers_grille(liste)` qui à partir d'une liste de N^2 chiffres, renvoie une grille (sous la forme d'une liste de listes).

Par exemple la liste :

```
[1, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 3, 0, 4]
```

devient la grille :

```
[[1, 0, 0, 0], [0, 0, 2, 0], [0, 0, 0, 0], [0, 3, 0, 4]]
```

Indication. Utilise la fonction `numero_vers_case()` après avoir initialisé une grille par

```
grille = [[0 for j in range(N)] for i in range(N)]
```

- 5. Chiffres en présence.** Programme des fonctions qui déterminent les chiffres déjà présents sur la grille :
- `chiffres_ligne(i, grille)` renvoie la liste des chiffres présents sur la ligne i .
 - `chiffres_colonne(j, grille)` renvoie la liste des chiffres présents sur la colonne j .
 - `chiffres_bloc(i, j, grille)` renvoie la liste des chiffres présents dans le même sous-bloc que la case (i, j) . (*Indication* : tu peux calculer les coordonnées (a, b) du coin supérieur gauche du sous-bloc par les formules : $a = M1 * (i // M1)$ et $b = Mc * (j // Mc)$.)
- 6. Choix.** Programme la fonction `choix(k)` qui renvoie la liste des chiffres jouables pour la case numéro k , en tenant compte des chiffres de la grille de départ et de ceux la configuration en cours de test (donnée par les chiffres dans les cases numéro 0 à $k - 1$).

Méthode. Tout d'abord le nombre total de piles est $n = N^2$. Puis pour la fonction `choix(k)` :

- si la case numéro k contient déjà un chiffre de la grille de départ, on le conserve, la liste à renvoyer contient uniquement ce chiffre, c'est terminé.
 - Sinon, il faut reconstituer une grille. Pour cela tu récupères d'abord la configuration en cours de test, par la commande `haut_des_piles()` qui contient la liste des chiffres des cases 0 à $k - 1$ (à convertir en une grille $N \times N$). Ensuite il ne faut pas oublier d'ajouter les chiffres de `grille_depart`.
 - Les chiffres à éviter pour la case k sont les chiffres présents sur la même ligne, ou la même colonne, ou dans le même sous-bloc. Les chiffres jouables sont les autres !
7. **Exemples.** Voici des exemples de sudokus de difficultés variées, sous la forme d'une liste de 64 nombres (à convertir en grille par la fonction `liste_vers_grille()`).

Facile

```
[
3,0,4, 0,8,0, 0,5,0,
7,0,0, 0,1,0, 0,0,3,
8,0,0, 0,0,2, 6,0,0,
0,0,9, 1,0,0, 3,0,5,
4,0,5, 3,0,7, 9,0,2,
6,0,8, 0,0,9, 7,0,0,
0,0,7, 4,0,0, 0,0,6,
5,0,0, 0,9,0, 0,0,8,
0,4,0, 0,7,0, 5,0,9,
]
```

Moyen

```
[
5,0,8, 0,3,0, 4,6,0,
0,0,0, 2,0,0, 8,0,0,
1,9,0, 4,0,0, 7,3,0,
8,0,7, 9,2,0, 0,0,0,
0,0,9, 6,0,4, 2,0,0,
0,0,0, 0,8,3, 1,0,5,
0,3,1, 0,0,2, 0,7,6,
0,0,2, 0,0,9, 0,0,0,
0,7,5, 0,6,0, 9,0,8,
]
```

Difficile

```
[
0,4,0, 1,0,0, 9,0,0,
0,0,0, 0,0,0, 0,6,0,
0,8,0, 6,3,0, 0,0,0,
5,1,7, 2,9,0, 0,0,8,
0,0,4, 0,5,0, 2,0,0,
9,0,0, 0,1,4, 7,5,6,
0,0,0, 0,7,5, 0,8,0,
0,9,0, 0,0,0, 0,0,0,
0,0,1, 0,0,2, 0,4,0,
]
```

Variable

```
[
8,1,2, 0,0,0, 0,0,0,
0,0,3, 6,0,0, 0,0,0,
0,7,0, 0,9,0, 2,0,0,
0,5,0, 0,0,7, 0,0,0,
0,0,0, 0,4,5, 7,0,0,
0,0,0, 1,0,0, 0,3,0,
0,0,1, 0,0,0, 0,6,8,
0,0,8, 5,0,0, 0,1,0,
0,9,0, 0,0,0, 4,0,0,
]
```

Cette dernière grille est déjà difficile (quelques secondes à résoudre), si on change la première ligne en :

```
8,1,0, 0,0,0, 0,0,0,
```

alors la résolution devient très difficile (plusieurs dizaines de secondes). Enfin, si on ne retient que le premier chiffre de la première ligne :

```
8,0,0, 0,0,0, 0,0,0,
```

notre programme échoue à trouver la solution en un temps raisonnable !

Fractale de Lyapunov

Dans une contrée lointaine, des loups se nourrissent des ressources de la région. Chaque année des loups naissent, d'autres meurent, parfois la population des loups reste stable d'une année sur l'autre. Mais d'autres fois la population augmente, ce qui fait qu'il n'y a plus assez de nourriture pour tous, ainsi beaucoup de loups meurent et l'année suivante il y a peu de loups. Mais alors les quelques loups qui restent disposent de beaucoup de nourriture et se reproduisent rapidement, la population augmente et bientôt il y a trop de loups...



Nous allons étudier des suites dont le comportement peut être chaotique. La fonction logarithme nous aidera à déterminer le caractère stable ou instable de la suite. Avec beaucoup de calculs et de patience nous tracerons des fractales très différentes de l'ensemble de Mandelbrot : les fractales de Lyapunov.

Cours 1 (La suite logistique).

La suite logistique est une suite mystérieuse définie par récurrence. On fixe d'abord un réel r avec $0 \leq r \leq 4$. Il y a une suite pour chaque paramètre r . La suite **suite logistique de paramètre r** est la suite $(u_n)_{n \in \mathbb{N}}$ définie par récurrence :

$$u_0 = \frac{1}{2} \quad \text{et} \quad u_{n+1} = r \cdot u_n \cdot (1 - u_n) \quad \text{pour } n \geq 0.$$

Exemples.

- Fixons $r = \frac{1}{2}$. Alors les premiers termes de la suite sont :

$$u_0 = \frac{1}{2} \quad u_1 = \frac{1}{2} \frac{1}{2} \left(1 - \frac{1}{2}\right) = \frac{1}{8} \quad u_2 = \frac{1}{2} \frac{1}{8} \left(1 - \frac{1}{8}\right) = \frac{7}{128}$$

$$u_3 = \frac{1}{2} \frac{7}{128} \left(1 - \frac{7}{128}\right) = 0.0258 \dots \quad u_4 = 0.0125 \dots$$

La suite semble tendre vers 0.

- Fixons $r = \frac{3}{2}$. Alors les premiers termes de la suite sont :

$$u_0 = \frac{1}{2} \quad u_1 = \frac{3}{2} \frac{1}{2} \left(1 - \frac{1}{2}\right) = \frac{3}{8} = 0.375 \quad u_2 = \frac{3}{2} \frac{3}{8} \left(1 - \frac{3}{8}\right) = 0.351 \dots$$

$$u_3 = 0.341 \dots \quad u_4 = 0.337 \dots$$

La suite semble tendre vers $\frac{1}{3}$.

- Fixons $r = 3.2$. Voici les premiers termes de la suite :

$$\begin{array}{ll} u_0 = 0.5 & u_1 = 0.8 \\ u_2 = 0.512 & u_3 = 0.79953 \dots \\ u_4 = 0.51288 \dots & u_5 = 0.79946 \dots \\ u_6 = 0.51301 \dots & u_7 = 0.799457 \dots \\ u_8 = 0.51304 \dots & u_9 = 0.799455 \dots \end{array}$$

La suite (u_n) n'a apparemment pas de limite, mais cependant les termes de rang pair u_{2n} tendent vers une valeur 0.51304..., alors que les termes de rang impair u_{2n+1} tendent vers une valeur 0.79945...

Les loups. La valeur de u_n représente la population de loups à l'année n . Si $u_n = 0$ il n'y a plus de loups, si $u_n = 1$ il y a le maximum de loups. Le paramètre r représente la vitesse de reproduction des loups (il ne change pas d'une année sur l'autre). La relation $u_{n+1} = r \cdot u_n \cdot (1 - u_n)$ implique en particulier que si u_n est proche de 0 ou proche de 1 alors u_{n+1} est proche de 0.

Reprenons les exemples un par un :

- $r = 0.5$. Les loups ne se reproduisent pas assez, la population finit par disparaître.
- $r = \frac{3}{2}$. La population de loups finit par se stabiliser.
- $r = 3.2$. La population ne se stabilise pas mais oscille entre deux valeurs d'une année sur l'autre.

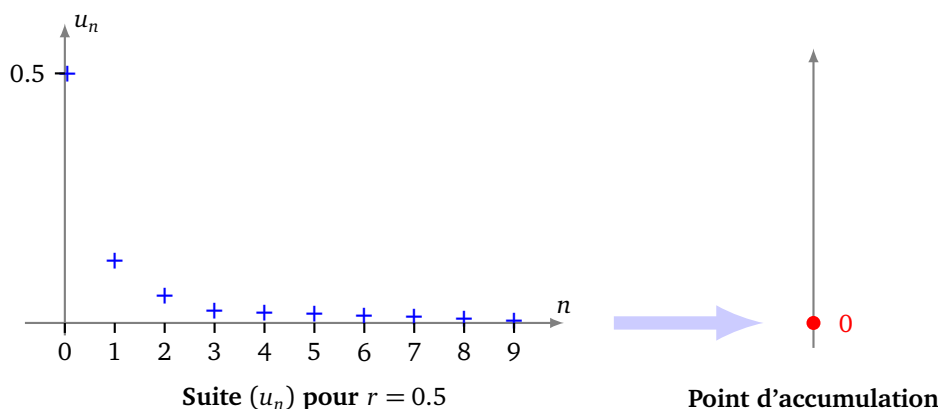
Cours 2 (Points d'accumulation).

Dans les exemples précédents on s'aperçoit que pour $r = 0.5$ ou bien $r = 1.5$ la suite (u_n) possède une limite. Par contre, pour $r = 3.2$ la suite $(u_n)_{n \in \mathbb{N}}$ n'a pas de limite. On pourrait presque dire que dans ce cas la suite (u_n) possède deux limites, mais on s'interdit cet usage, car si une limite existe elle doit être unique.

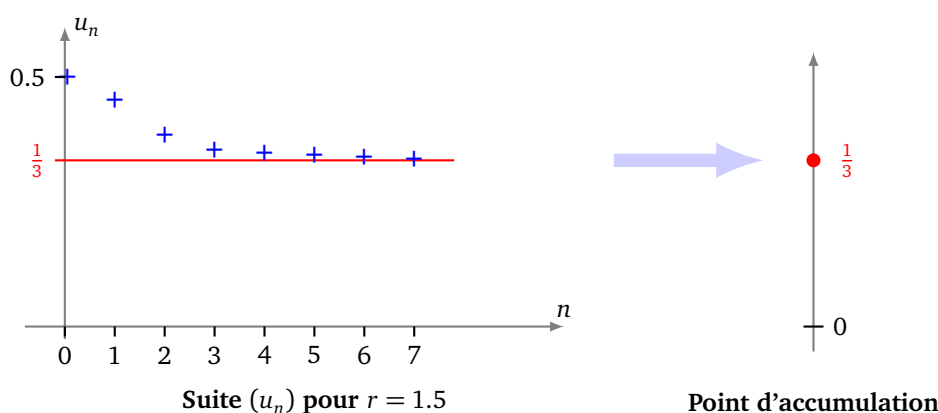
Dans le cas $r = 3.2$ on dit que la suite (u_n) possède deux **points d'accumulation** (ce sont les valeurs « limites » 0.51304... et 0.79945...).

Voici comment on va obtenir une approximation des points d'accumulation. On calcule les premiers termes de la suite (u_n) mais on ne retient que les termes $u_{100}, u_{101}, \dots, u_{199}$. Ces valeurs donnent une bonne approximation des points d'accumulation possibles.

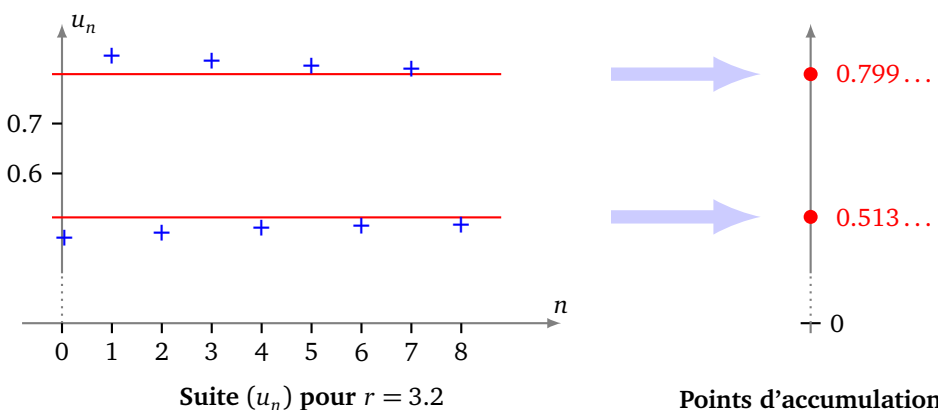
Dans le cas $r = 0.5$, la suite (u_n) tend très vite vers 0, ainsi tous les termes de u_{100} , à u_{199} sont quasiment nuls. Ils approchent donc bien la limite qui est $\ell = 0$ (c'est le seul point d'accumulation).



Dans le cas $r = 1.5$, la suite (u_n) tend très vite vers $\frac{1}{3}$, ainsi tous les termes de u_{100} à u_{199} approchent bien la limite qui est $\ell = \frac{1}{3}$.



Dans le cas $r = 3.2$, les termes u_{100} à u_{199} alternent entre des valeurs proches de chacun des deux points d'accumulation $\ell_1 = 0.51304\dots$ et $\ell_2 = 0.79945\dots$.

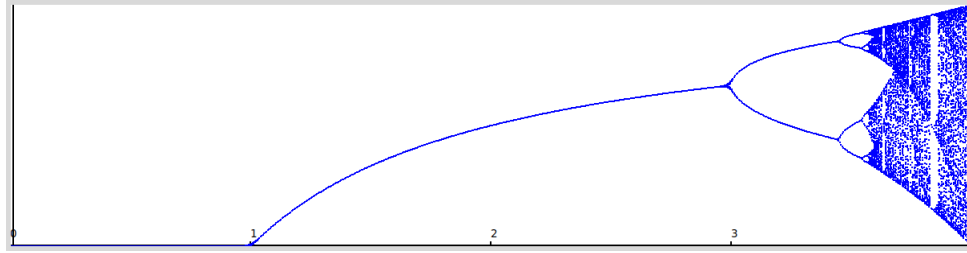


Dans le cas $r = 3.5$, la suite possède quatre points d'accumulation $\ell_1 = 0.50088\dots$, $\ell_2 = 0.87499\dots$, $\ell_3 = 0.38281\dots$ et $\ell_4 = 0.82694\dots$.

Conclusion : pour trouver une valeur approchée d'une limite ou des points d'accumulation, on se contente des valeurs de la suite de u_{100} à u_{199} .

Activité 1.

Objectifs : tracer les points d'accumulation de la suite logistique.



On rappelle que la suite logistique est définie pour un paramètre $0 \leq r \leq 4$ par :

$$u_0 = \frac{1}{2} \quad \text{et} \quad u_{n+1} = r \cdot u_n \cdot (1 - u_n) \quad \text{pour } n \geq 0.$$

1. Programme une fonction `liste_suite(r, Nmin, Nmax)` qui renvoie la liste des termes de la suite logistique u_n pour n variant de $N_{\min}$ à $N_{\max} - 1$.

Par exemple `liste_suite(1.5, 0, 5)` renvoie les 5 premiers termes de la suite définie par le paramètre $r = 1.5$: $[0.5, 0.375, 0.35156..., 0.34194..., 0.33753...]$.

2. Programme une fonction `bifurcation(Nmin, Nmax, epsilon)` qui renvoie la liste des points (r, u_n) où :

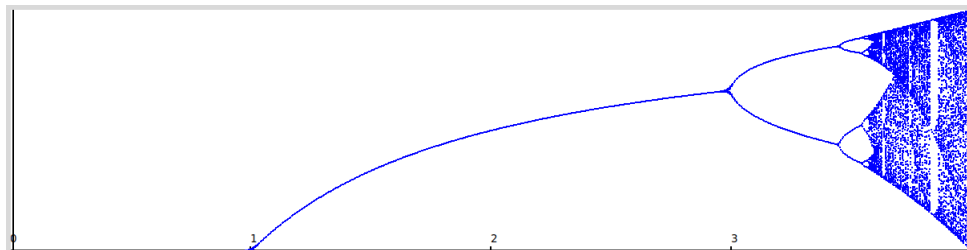
- r est un paramètre réel qui varie de 0 à 4, par saut de longueur ϵ ,
- n est un entier qui varie de $N_{\min}$ à $N_{\max} - 1$,
- u_n est le terme de la suite logistique de paramètre r .

Par exemple avec $N_{\min} = 0$ à $N_{\max} = 5$ et $\epsilon = 1$, la fonction `bifurcation()` renvoie la liste des points :

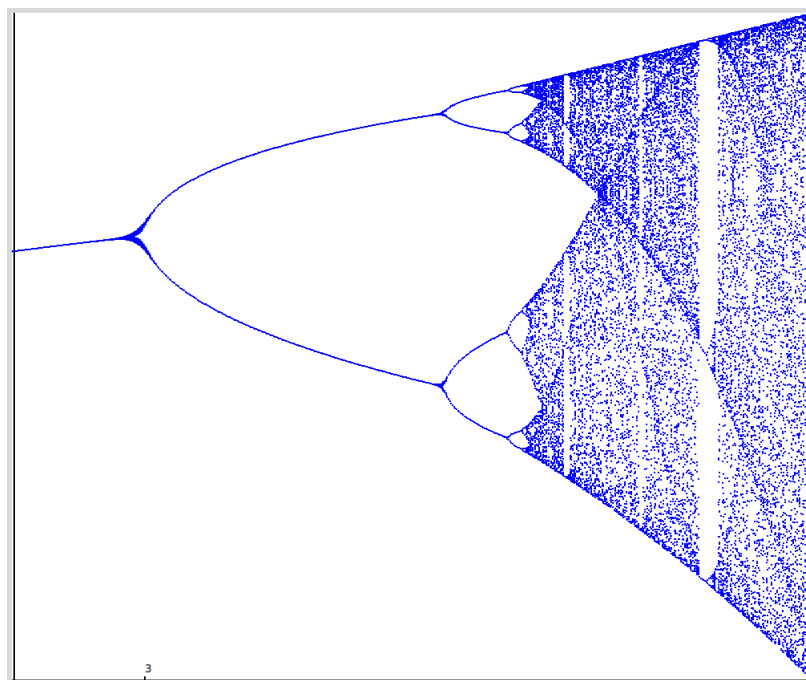
```
[(0, 0.5), (0, 0.0), (0, 0.0), (0, 0.0), (0, 0.0),
(1, 0.5), (1, 0.25), (1, 0.1875), (1, 0.15234375), (1, 0.1291351318359375),
(2, 0.5), (2, 0.5), (2, 0.5), (2, 0.5), (2, 0.5),
(3, 0.5), (3, 0.75), (3, 0.5625), (3, 0.73828125), (3, 0.5796661376953125),
(4, 0.5), (4, 1.0), (4, 0.0), (4, 0.0), (4, 0.0)]
```

correspondant aux premiers termes des suites pour $r = 0, 1, 2, 3, 4$.

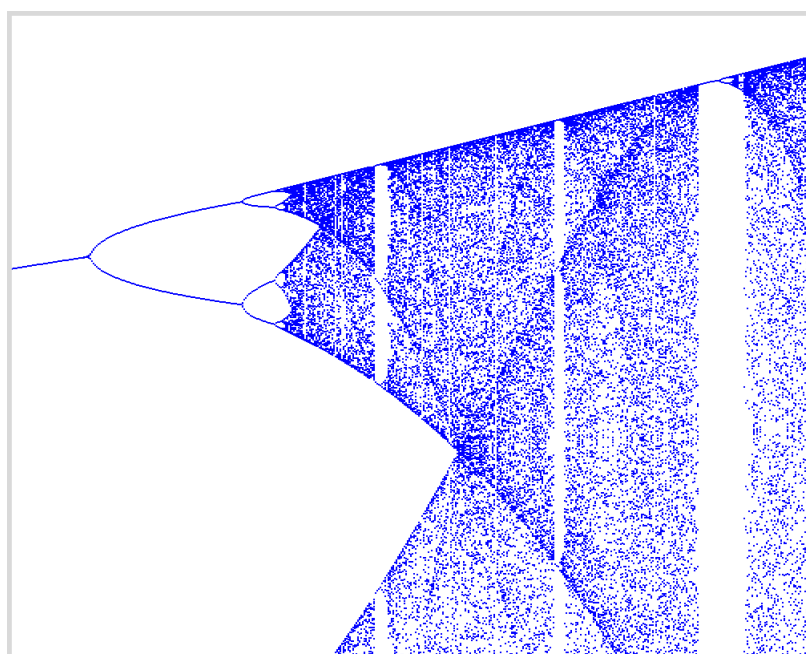
3. Affiche les points d'accumulation approchés (r, u_n) pour obtenir le diagramme de bifurcation. On prendra $N_{\min} = 100$ à $N_{\max} = 200$ et $\epsilon = 0.01$. Pour obtenir plus de points, on diminuera la valeur de ϵ .



L'axe horizontal correspond aux valeurs de $r \in [0, 4]$. L'axe vertical correspond aux valeurs de $u_n \in [0, 1]$. Chaque point (r, u_n) approche un point d'accumulation de la suite (u_n) de paramètre r .



Zone où $r \geq 2.8$.



Zoom sur la zone où $r \in [3.3, 3.9]$ et $u_n \in [0.6, 1]$.

Cours 3 (L'exposant de Lyapunov).

L'exposant de Lyapunov mesure la stabilité de nos suites logistiques. Plus l'exposant est négatif plus la suite est stable, plus l'exposant est positif plus la suite est chaotique.

Définition générale.

Considérons une suite $(u_n)_{n \in \mathbb{N}}$ définie par un terme initial et une formule de récurrence

$$u_0 \in \mathbb{R} \quad \text{et} \quad u_{n+1} = f(u_n)$$

où $f : \mathbb{R} \rightarrow \mathbb{R}$ est une fonction.

L'**exposant de Lyapunov** associé à cette suite est :

$$L = \lim_{N \rightarrow +\infty} \frac{1}{N} \sum_{n=0}^{N-1} \ln(|f'(u_n)|)$$

C'est une moyenne de logarithmes. On va expliquer plus bas comment calculer cette formule compliquée dans notre cas.

L'exposant de Lyapunov approché pour la suite logistique.

Dans notre cas la suite logistique est définie à l'aide de la fonction $f(x) = rx(1-x)$, pour laquelle $f'(x) = r - 2rx$, de sorte que pour N assez grand :

$$L_r \simeq \frac{1}{N} \sum_{n=0}^{N-1} \ln(|r - 2ru_n|)$$

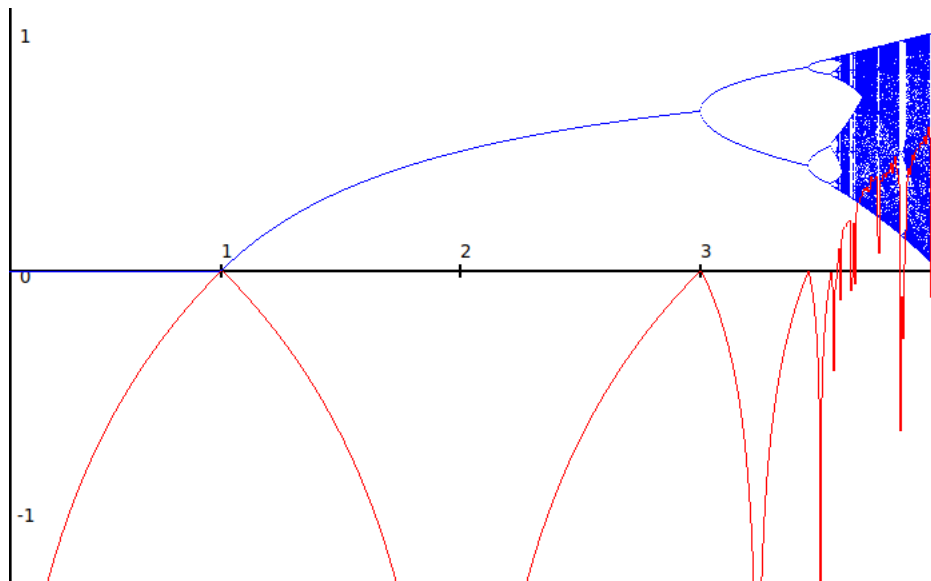
c'est-à-dire

$$L_r \simeq \frac{1}{N} (\ln(|r - 2ru_0|) + \ln(|r - 2ru_1|) + \dots + \ln(|r - 2ru_n|) + \dots + \ln(|r - 2ru_{N-1}|))$$

Attention ! Il faut supposer $r > 0$. Il faut aussi exclure de cette somme les termes $u_n = \frac{1}{2}$ (car le logarithme n'est pas défini en 0).

Activité 2 (Exposant de Lyapunov).

Objectifs : calculer l'exposant de Lyapunov et le visualiser.



1. Programme une fonction `exposant_lyapunov(liste_u, r)` qui renvoie une valeur approchée de l'exposant de Lyapunov selon la formule :

$$L_r \simeq \frac{1}{N} \sum_{u \in \text{liste}} \ln(|r - 2ru|)$$

où N est le nombre d'éléments de la liste.

Indication. N'oublie pas d'exclure de la somme les cas où $u = \frac{1}{2}$.

2. Programme une fonction `bifurcation_lyapunov()` qui affiche le graphe de la fonction $r \mapsto L_r$.

Indications.

- Voir le tracé (en rouge) sur la figure ci-dessus.

- Autrement dit, il faut afficher les points (r, L_r) pour r variant de 0 à 4 par exemple.
- Il vaut mieux ici relier les points correspondant (r, L_r) à $(r + \epsilon, L_{r+\epsilon})$, avec un petit pas ϵ , pour obtenir un plus joli tracé.
- C'est encore mieux de tracer ce graphe sur le même graphique que le diagramme de bifurcation.
- Pour approcher efficacement la limite dans la formule de l'exposant de Lyapunov il est conseillé de ne retenir que les termes de rang $N_{\min}$ à $N_{\max} - 1$, avec par exemple comme auparavant $N_{\min} = 100$ et $N_{\max} = 200$.

Exemples.

- $r = 0.5$. Si on calcule l'exposant de Lyapunov en se limitant aux premiers termes $[u_0, u_1, \dots, u_9]$ alors on trouve $L_{0.5} \simeq -0.67$. Pour une meilleure approximation, on calcule la somme avec les termes u_{100} à u_{199} (donc avec $N_{\min} = 100$ à $N_{\max} = 200$), on trouve $L_{0.5} \simeq -0.69$. Un exposant négatif correspond à une situation stable : ici la suite tend vers 0.
- $r = 3$. On trouve $L_3 = 0$. L'exposant est nul, la situation n'est pas très stable (pour $r < 3$ la suite a une limite, pour r juste plus grand que 3 la suite possède deux points d'accumulation).
- $r = 3.2$. On trouve $L_{3.2} \simeq -0.91$. C'est une situation stable, même si la suite ne converge pas ses valeurs oscillent régulièrement entre deux valeurs limites.
- $r = 3.7$. On trouve $L_{3.7} \simeq 0.32$. L'exposant est positif, nous sommes dans la zone chaotique.

Cours 4 (Suite logistique suivant un motif).

Motif.

- Un **motif** est une suite de lettres **A** ou **B**, par exemple **AB** ou **AABA**.
- En répétant un motif, on obtient un mot de longueur infinie, par exemple avec **AB** on obtient **ABABABABAB...** et avec **AABA** on obtient **AABAABAABAABA...**
- La n -ème lettre tirée d'un motif est la n -ème lettre du mot infini issu du motif (en commençant avec $n = 0$). Par exemple pour le motif **AB**, pour n pair la lettre est **A** et n impair la lettre est **B**. Pour le motif **AABA** et $n = 6$ la lettre est **B**.

Suite logistique d'après un motif. On va définir une suite logistique à deux paramètres.

- On fixe deux valeurs réelles r_1 et r_2 dans l'intervalle $[0, 4]$.
- On fixe un motif.
- On initialise une suite à $u_0 = \frac{1}{2}$.
- La suite logistique de paramètres (r_1, r_2) est définie par récurrence :
 - si la n -ème lettre issue du motif est **A** alors $u_{n+1} = r_1 u_n (1 - u_n)$,
 - si la n -ème lettre issue du motif est **B** alors $u_{n+1} = r_2 u_n (1 - u_n)$.

Exemple. Calculons la suite logistique avec le motif **AB** et les paramètres $(\frac{1}{2}, \frac{7}{2}) = (0.5, 3.5)$. Le motif donne le mot **ABABABABAB...** on va donc alterner le paramètre r_1 et le paramètre r_2 .

- $u_0 = \frac{1}{2}$ (terme initial),
- $n = 0$, la lettre de rang 0 est **A**, donc $u_1 = r_1 u_0 (1 - u_0) = \frac{1}{8} = 0.125$.
- $n = 1$, la lettre de rang 1 est **B**, donc $u_2 = r_2 u_1 (1 - u_1) = 0.382 \dots$
- $n = 2$, la lettre de rang 2 est **A**, donc $u_3 = r_1 u_2 (1 - u_2) = 0.118 \dots$
- $u_4 = r_2 u_3 (1 - u_3) = 0.364 \dots$
- $u_5 = 0.115 \dots$

Cours 5 (Fractale de Lyapunov).

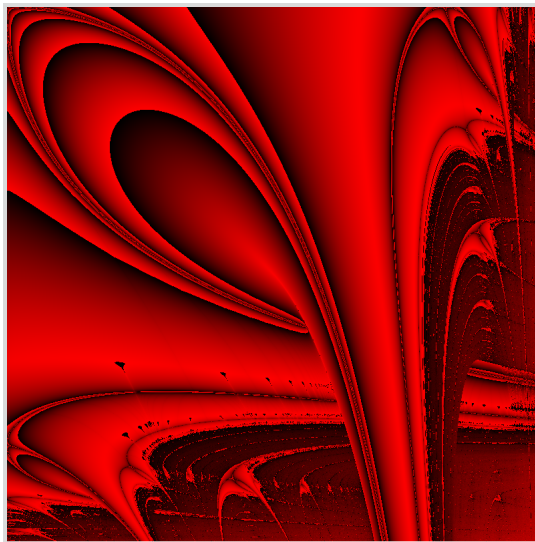
Voici comment est définie une fractale de Lyapunov :

- Tout d'abord on fixe un motif.
- La fractale est dessinée dans la zone $[0, 4] \times [0, 4]$, correspondant aux paramètres (r_1, r_2) .
- Pour chacun des paramètres (r_1, r_2) :
 - on calcule la suite logistique associée à ces paramètres et au motif,
 - on calcule l'exposant de Lyapunov de cette suite,
 - on colorie le pixel (r_1, r_2) en fonction de la valeur de l'exposant.

Si on change de motif on obtient une fractale un peu différente !

Activité 3 (Fractale de Lyapunov).

Tracer les fractales de Lyapunov. Il est bon d'avoir déjà tracé la fractale de Mandelbrot, le principe étant similaire, mais ici les calculs sont beaucoup plus lents.



1. Programme une fonction `A_ou_B(motif, n)` qui pour une chaîne `motif`, renvoie le caractère 'A' ou 'B' de rang n du mot itéré.

Exemple. `A_ou_B('AB', 10)` renvoie 'A' alors que `A_ou_B('AAABBB', 123)` renvoie 'B'.

2. Modifie ta fonction précédente en une fonction `liste_suite_motif(r1, r2, motif, Nmin, Nmax)` qui renvoie la suite logistique construite d'après le motif donné et les paramètres (r_1, r_2) sous la forme d'une liste de termes u_n avec $N_{\min} \leq n < N_{\max}$.

Exemple. Avec $r_1 = 0.5$ et $r_2 = 3.5$ et le motif **AB** alors la commande `liste_suite_motif(r1, r2, motif, 0, 10)` renvoie les dix premiers termes de la suite logistique :

[0.5, 0.125, -1.0937..., -0.273..., 2.392...,
0.598..., -5.233..., -1.308..., 11.448..., 2.862...]

3. Modifie ta fonction précédente en une fonction

`exposant_lyapunov_motif(r1, r2, motif, Nmin, Nmax)`

qui calcule l'exposant de Lyapunov de la liste des termes de la suite logistique avec $N_{\min} \leq n < N_{\max}$.

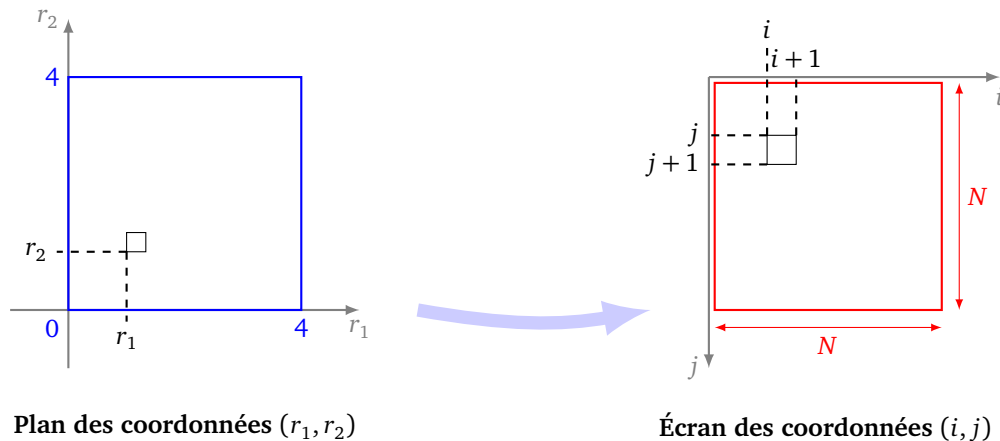
Indications. Reprends une partie du code de la fonction `exposant_lyapunov()` que tu intègres dans le code de ta fonction `liste_suite_motif()`.

Exemple. Prenons $r_1 = 0.5$ et $r_2 = 3.5$ et le motif **AB** alors, avec les 10 premiers termes de la suite ($N_{\min} = 0, N_{\max} = 10$), on trouve un exposant de Lyapunov de $-0.506\dots$ Pour obtenir la valeur à la limite qui donne l'exposant, on prend en compte plus de termes, après avoir oublié les premiers termes, par exemple en posant $N_{\min} = 100, N_{\max} = 200$. On obtient alors une bonne approximation de l'exposant de Lyapunov de la suite logistique $L_{r_1, r_2} = -0.940147\dots$

4. Programme une fonction `fractale_lyapunov(motif)` qui dessine la fractale de Lyapunov associée au motif.

Suis l'algorithme suivant :

- Fixer la taille de la fenêtre N en pixel ($N = 100$ pour commencer car les calculs sont très longs).
- La fenêtre réelle étant la zone $[0, 4] \times [0, 4]$, fixer un pas h défini par $h = 4/N$.
- Initialiser r_1 à 0.
- Pour i allant de 0 à N :
 - Initialiser r_2 à 0.
 - Pour j allant de 0 à N :
 - calculer l'exposant de Lyapunov de la suite logistique associée au motif donné et aux paramètres (r_1, r_2) ,
 - colorier le pixel (i, j) en fonction de cet exposant,
 - faire $r_2 \leftarrow r_2 + h$.
 - Faire $r_1 \leftarrow r_1 + h$.



Couleurs. Voici une fonction qui renvoie une couleur en fonction d'une valeur ℓ de l'exposant de Lyapunov.

```
def choix_couleur(l):
    i = round(150*l)
    R,V,B = i,0,0      # Nuances de rouge
    couleur = '#%02x%02x%02x' % (R%256, V%256, B%256)
    return couleur
```

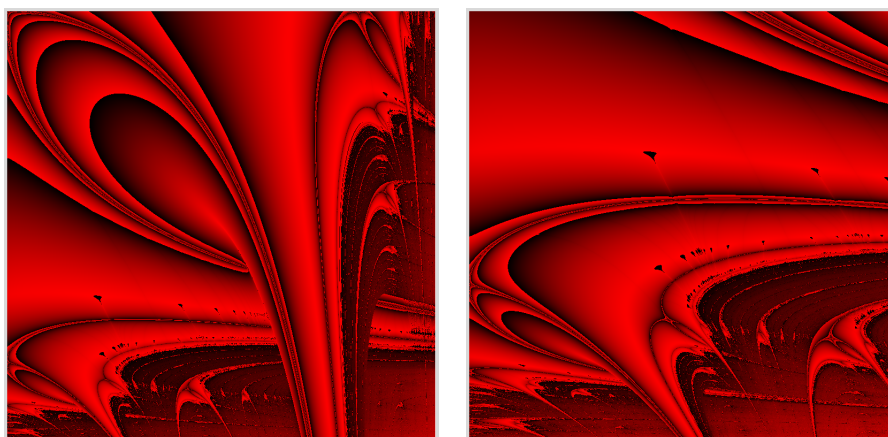
Zoom. Pour afficher une fenêtre précise correspondant à une zone plus petite que la zone $[0, 4] \times [0, 4]$ voir l'activité « Ensemble de Mandelbrot ».

De gauche à droite les fractales de Lyapunov pour les motifs **AB**, **AAABB**, **AAAAABABBB**.



Il est amusant d'explorer différents motifs, de changer la formule pour les couleurs et de faire des zooms sur les parties chaotiques.

Voici des zooms pour le motif **AB**, à gauche sur la zone $[2, 4] \times [2, 4]$, à droite sur la zone $[2, 3] \times [3, 4]$.



Big data I

Big data, intelligence artificielle, deep learning, réseau de neurones, machine learning... plein de mots compliqués ! Le but commun est de faire exécuter à un ordinateur des tâches de plus en plus complexes : choisir (par exemple trouver un bon élément parmi des milliards selon plusieurs critères), décider (séparer des photos de chats de photos de voitures), prévoir (un malade a de la fièvre et le nez qui coule, quelle maladie est la plus probable ?). Dans cette première partie, on va utiliser des outils classiques de statistique et de probabilité pour résoudre des problèmes amusants.

Cours 1 (Des données par milliers).

Lorsque l'on parle de *big data* on parle de fichiers avec des milliards de données. On va plus modestement traiter de fichiers contenant les données (fictives) de 100 000 personnes.

Récupères le fichier `personnes_100000.csv`. Tu disposes aussi des fichiers `personnes_100.csv`, `personnes_1000.csv`, `personnes_10000.csv` qui contiennent moins d'entrées et sont idéaux pour tester les programmes.

Ces fichiers sont disponibles ici :

github.com/exo7math/python2-exo7

Voici un exemple de ligne de ce fichier au format *csv* (*comma separated values*) :

femme, Lessard, Capucine, 7/31/1978, LA ROCHELLE, 56.7, 155, 0+

ou

homme, Cadieux, Antoine, 11/20/1938, METZ, 78.8, 166, A-

Chaque ligne désigne une personne et ses caractéristiques :

- sexe (homme/femme)
- nom
- prénom
- date de naissance (format jj/mm/aaaa)
- ville
- poids (en kilogrammes)
- taille (en centimètres)
- groupe sanguin

Activité 1 (Sondage).

Objectifs : utiliser un échantillon pour déterminer les caractéristiques d'une population.

1. Programme une fonction `age_moyen(debut, fin, fichier)` qui évalue l'âge moyen d'une liste de personnes (contenue dans le fichier donné) à partir d'un échantillon. Par exemple :

`age_moyen(10, 20, "personnes_100.csv")` renvoie la moyenne des âges des personnes de rang 10 (compris) à 20 (exclu) parmi une liste de 100 personnes.

Indication. L'âge d'une personne est l'âge qu'elle aura à la fin de l'année en cours.

Compare l'âge moyen calculé à partir de l'échantillon avec l'âge moyen de toute la liste. Quelle taille de l'échantillon permet d'avoir une estimation à 1 an près ?

2. Programme une fonction `probabilite_initiale(lettre, debut, fin, fichier)` qui estime la probabilité que le nom d'une personne commence par la lettre donnée à partir d'un échantillon de la liste. Pour cela on approche la probabilité par la formule :

$$\text{probabilité} \simeq \frac{\text{nombre d'occurrences}}{\text{nombre total d'éléments}}$$

3. Programme une fonction `probabilite_groupe_sanguin(debut, fin, fichier)` qui renvoie les probabilités de chaque groupe sanguin. La fonction renvoie un dictionnaire dont le couple « clé/valeur » est « groupe sanguin/probabilité ». Les groupes possibles sont A+, A-, B+, B-, O+, O-, AB+ et AB-. Par exemple (obtenu sur un échantillon de 10 personnes seulement) :

```
{ 'A+': 0.1, 'A-': 0.1, 'B+': 0.1, 'B-': 0.1,
  'O+': 0.3, 'O-': 0.3, 'AB+': 0.0, 'AB-': 0.0 }
```

Quels sont les groupes les plus fréquents ?

Cours 2 (Ordre alphabétique).

On rappelle que Python connaît l'ordre alphabétique et qu'on peut comparer deux chaînes de caractères :

- « "A" < "B" » est « Vrai » (Python renvoie True),
- « "BAC" < "ABC" » est « Faux »,
- « "A" == "a" » est « Faux ».

Tu peux consulter la fiche « Le mot le plus long » et aussi la fiche « Tri - Complexité » pour plus d'informations et une activité similaire à l'activité suivante.

Activité 2 (Chercher dans une liste de noms).

Objectifs : chercher dans une liste élément par élément ou bien par dichotomie.

1. **Liste triée.** Programme une fonction `fichier_vers_liste_noms(fichier)` qui renvoie la liste ordonnée des noms issus du fichier.

Indication. `liste.sort()` trie la liste.

2. **Début d'une chaîne.** Programme une fonction `est_debut(debut, chaine)` qui teste si une chaîne débute par les caractères donnés. Par exemple :

- `est_debut("ABC", "ABCDEF")` renvoie « Vrai »,
- `est_debut("XYZ", "ABCDEF")` renvoie « Faux »,
- `est_debut("ABCD", "AB")` renvoie « Faux ».

3. **Recherche séquentielle.** Programme une fonction `chercher_1(liste, debut)` qui renvoie un nom de la liste commençant par `debut` (ou None si un tel nom n'existe pas).

Méthode. Parcoure un par un les noms de la liste (issue de la première question) et teste s'il commence par `debut`.

Par exemple `chercher_1(liste, "Bri")` peut renvoyer 'Brian'.

4. **Recherche dichotomique.** Programme une fonction `chercher_2(liste, debut)` qui fait le même travail mais avec un algorithme plus efficace : la dichotomie.

Algorithme.

- — Entrée : un début de mot à trouver et une liste ordonnée de noms.
- — Sortie : un mot trouvé dans la liste commençant par le début souhaité ou None en cas d'échec.
- $a \leftarrow 0$ (le rang d'une liste commence à 0).
- $b \leftarrow n - 1$ où n est la longueur de la liste.
- Tant que $b \geq a$, faire :
 - $k \leftarrow (a + b) // 2$
 - Si debut est le début du mot `liste[k]` alors renvoyer `liste[k]`.
 - Si debut vient après `liste[k]` dans l'ordre alphabétique alors faire $a \leftarrow k + 1$,
 - sinon faire $b \leftarrow k - 1$.
- Renvoyer None (c'est le cas uniquement si aucun nom n'a été trouvé dans la boucle précédente).

Par exemple `chercher_2(liste, "Bri")` peut renvoyer 'Brisebois'. Le nom trouvé par cet algorithme n'est pas nécessairement le premier qui viendrait dans l'ordre alphabétique.

5. Complexité.

- (a) Modifie tes deux fonctions de recherche pour qu'elles renvoient en plus du nom, le nombre d'itérations nécessaires (par exemple le nombre de fois où tu effectues un appel à la fonction `est_debut()`).
- (b) Vérifie expérimentalement que si n est la longueur de la liste ordonnée, alors :
 - pour la recherche séquentielle, il peut y avoir jusqu'à n itérations,
 - pour la recherche par dichotomie, il peut y avoir jusqu'à $E(\log_2(n) + 1)$ itérations (où $E(x)$ désigne la partie entière, comme la commande `floor(x)`).

Activité 3 (La formule des tanks).

Objectifs : déterminer la taille N d'une série $1, \dots, N$ en ne connaissant que quelques numéros tirés au hasard.

En plein milieu de la seconde guerre mondiale les Allemands produisent un nouveau tank plus performant. Les Alliés s'inquiètent car ils ne savent pas combien de ces nouveaux tanks sont produits. Les services de renseignements estiment la production à 1500 tanks par mois. Que disent les mathématiques ? Les Alliés ont intercepté 4 tanks produits le même mois et qui portent les numéros :

143 77 198 32

Combien de tanks ont été produit ce mois ?

Modélisation. Sachant que les tanks sont numérotés de 1 à N chaque mois, à quelle valeur peut être estimée la production mensuelle N , connaissant un échantillon de k numéros $[n_1, n_2, \dots, n_k]$?

1. **La formule des tanks.** On note m le maximum des éléments de l'échantillon. On note k la taille de l'échantillon. Alors la formule des tanks estime :

$$N \simeq m + \frac{m}{k} - 1$$

Programme cette formule en une fonction `formule_tanks(echantillon)` qui renvoie cette estimation de N . Quelle est ton estimation pour le nombre de tanks ?

2. **Le double de la moyenne.** On peut essayer d'autres estimations. Par exemple on peut estimer N comme le double de la moyenne de l'échantillon. Programme une fonction

`double_moyenne(echantillon)` qui renvoie cette nouvelle estimation. Compare avec la formule des tanks.

Pour tester l'efficacité de la formule des tanks on va faire le cheminement inverse : on fixe un entier N , on choisit au hasard un échantillon de k éléments et on regarde si nos formules permettent de bien approcher N .

3. **Tirage sans remise.** Programme une fonction `tirage_sans_remise(N,k)` qui renvoie une liste de k entiers différents compris entre 1 et N (inclus).
4. **Erreurs.** Programme une fonction `erreurs(N,k)` (ou mieux `erreurs(N,k,nb_tirages=1000)`) qui calcule l'erreur moyenne commise par nos formules. Pour cela :
 - Effectue un tirage sans remise de k entiers plus petits que N .
 - Calcule la valeur N_1 obtenue à partir de cet échantillon par la formule des tanks.
 - Calcule l'erreur commise $e = |N - N_1|$.

En faisant ceci pour un grand nombre de tirages, calcule l'erreur moyenne commise. Fais le même travail avec l'autre formule et renvoie les deux erreurs moyennes.

Pour 20 entiers plus petits que 1000 ($k = 20$ et $N = 1000$) quelle est la meilleure formule et à quelle erreur peut-on s'attendre ?

À la fin de la guerre les registres Allemands ont été récupérés et indiquaient une production de 245 chars mensuels ! Cette formule est aussi utilisée pour estimer la production d'un produit (par exemple d'un téléphone) à partir des numéros de série.

Cours 3 (Le problème du secrétaire).

La directrice d'une entreprise doit choisir son nouveau secrétaire : elle reçoit $k = 100$ secrétaires un par un et attribue à chacun une note (par exemple un entier entre 0 et $N = 100$). Elle veut choisir le meilleur secrétaire mais il y a une contrainte : elle décide immédiatement après chaque entretien si elle embauche ou pas ce secrétaire. Elle n'a pas la possibilité de revenir en arrière.

Voici la stratégie qu'elle adopte : elle commence par recevoir un certain nombre de candidats (par exemple 25), elle mémorise juste la meilleure note obtenue jusqu'ici sans retenir aucun de ces candidats. Ensuite elle reçoit les candidats suivants et elle sélectionne le premier qui a une note supérieure ou égale à la meilleure note de l'échantillon. Avec cette stratégie elle n'est pas sûre de trouver le meilleur secrétaire, et elle peut même ne sélectionner aucun secrétaire.

Exemple. Voici une liste de scores des candidats (ici avec seulement 10 candidats) :

`liste = [2,5,3,4,1,6,4,5,8,3]`

Prenons le pourcentage $p = 25$. La taille de la liste est $k = 10$ donc l'échantillon de 25% est formé des 2 premiers éléments $[2,5]$. Le score maximum de cet échantillon est $M = 5$. La directrice ne retient pas de candidat dans l'échantillon, par contre elle va choisir le premier candidat suivant dont la note sera supérieure ou égale à M et arrête le processus. Ici c'est donc le candidat avec un score de 6 qui est choisi. Note qu'elle n'a pas sélectionné le meilleur candidat avec un score de 8 mais qui était en fin de liste. Le but est de choisir la bonne taille pour l'échantillon : avec un échantillon trop petit elle va choisir un secrétaire moyen, avec un échantillon trop grand elle ne va pas trouver de secrétaire du tout.

Notations.

- Le nombre de candidats secrétaires est k . Par défaut $k = 100$.
- Les notes vont de 0 à N . Par défaut $N = 100$.

- La taille de l'échantillon s'exprime comme un pourcentage p du nombre total de candidats k total. Par exemple $p = 25$ signifie que l'on teste d'abord 25% de candidats.

Activité 4 (Le problème du secrétaire).

Objectifs : programmer la sélection d'un bon secrétaire et optimiser la taille de l'échantillon.

1. Programme une fonction `genere_liste(k,N)` qui génère une liste de k entiers tirés au hasard entre 0 et N (deux nombres peuvent être identiques). Cela correspond aux notes des k secrétaires.
2. Programme une fonction `choix_secretaire(liste,p)` qui à partir d'une liste de notes et un pourcentage renvoie le score du secrétaire choisi (ou `None` si aucun ne convient). La méthode adoptée est celle décrite dans le cours au-dessus :
 - à partir de l'échantillon formé par les premiers candidats (la taille de l'échantillon est donnée sous la forme d'un pourcentage p) on retient le meilleur score M de cet échantillon, mais aucun n'est sélectionné,
 - en recevant un par un les candidats suivants, on prend le premier ayant un score supérieur ou égal à M ,
 - si aucun ne convient on renvoie `None`.
3. On souhaite savoir si notre stratégie est efficace ou pas : combien de fois sélectionne-t-on le meilleur secrétaire possible ? Pour cela tu vas tester la méthode sur un grand nombre de tirages.

Programme une fonction `meilleurs_secretaires(k,N,p,nb_tirages)` qui renvoie le nombre de fois où l'algorithme de la directrice sélectionne le meilleur candidat.

- k est la longueur des listes,
- chaque note est un entier entre 0 et N (inclus),
- p est le pourcentage qui détermine la taille de l'échantillon,
- `nb_tirages` est le (grand) nombre de listes aléatoires à tester.

Par exemple avec $k = 100$, $N = 100$ et le pourcentage $p = 25$ en effectuant de nombreux tirages (au moins 1000) le candidat sélectionné est dans environ 47% des cas le meilleur des candidats de toute la liste.

4. Le pourcentage de 25% pour l'échantillon n'est pas celui qui conduit aux meilleurs résultats. Écris un programme ou bien tâtonne pour trouver la meilleure valeur de p possible.

Indice. La réponse s'appelle la loi en $1/e$!

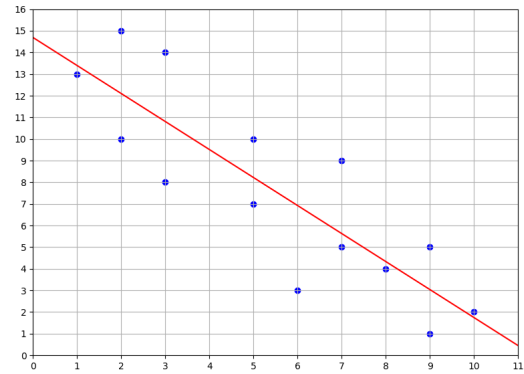
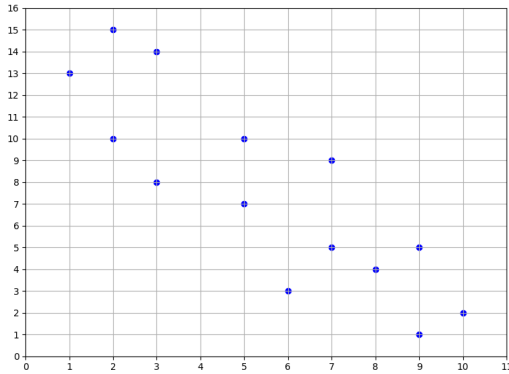
Cours 4 (Régression linéaire).

On se donne des points $(x_i, y_i)_{1 \leq i \leq n}$. On cherche si on peut modéliser la situation par une relation linéaire entre l'abscisse et l'ordonnée du type :

$$y = ax + b$$

C'est-à-dire que l'on cherche la droite qui « approche » au mieux tous les points. Cette opération s'appelle la **régression linéaire** et la droite est la **droite des moindres carrés**.

Sur la figure ci-dessous à gauche une série de points (x_i, y_i) , à droite la droite des moindres carrés d'équation $y = ax + b$.



Voici comment calculer les coefficients a et b de la droite $y = ax + b$.

- On note m_x la moyenne des $(x_i)_{1 \leq i \leq n}$.
- On note m_y la moyenne des $(y_i)_{1 \leq i \leq n}$.
- On note $\text{Var}(x)$ la **variance** des $(x_i)_{1 \leq i \leq n}$:

$$\text{Var}(x) = \frac{1}{n} \sum_{i=1}^n (x_i - m_x)^2$$

c'est-à-dire :

$$\text{Var}(x) = \frac{1}{n} ((x_1 - m_x)^2 + (x_2 - m_x)^2 + \dots + (x_n - m_x)^2)$$

La variance mesure l'écart des valeurs avec la moyenne. La variance est aussi le carré de l'écart-type.

- On note $\text{Cov}(x, y)$ la **covariance** des $(x_i)_{1 \leq i \leq n}$ avec $(y_i)_{1 \leq i \leq n}$:

$$\text{Cov}(x, y) = \frac{1}{n} \sum_{i=1}^n (x_i - m_x)(y_i - m_y)$$

c'est-à-dire :

$$\text{Cov}(x, y) = \frac{1}{n} ((x_1 - m_x)(y_1 - m_y) + (x_2 - m_x)(y_2 - m_y) + \dots + (x_n - m_x)(y_n - m_y))$$

La covariance mesure la corrélation (c'est-à-dire la dépendance) entre les valeurs x_i et y_i .

- Alors les coefficients de la droite $y = ax + b$ sont :

$$a = \frac{\text{Cov}(x, y)}{\text{Var}(x)} \quad \text{et} \quad b = m_y - am_x$$

Activité 5 (Régression linéaire).

Objectifs : tracer la droite des moindres carrés.

1. Moyenne, variance, covariance.

Programme :

- une fonction `moyenne(liste)` qui renvoie la moyenne des éléments (x_i) de la liste,
- une fonction `variance(liste)` qui renvoie la variance des éléments (x_i) de la liste,
- une fonction `covariance(listex, listey)` qui renvoie la covariance des éléments (x_i) et (y_i) .

Exemples.

- Vérifie que la moyenne de $(1, 2, 3, 4, 5)$ est $m_x = 3$ et la variance $\text{Var}(x) = 2$.
- Vérifie que la covariance entre $(1, 2, 3, 4, 5)$ et $(4, 5, 4, 7, 6)$ vaut $\text{Cov}(x, y) = 1.2$.
- Vérifie sur un exemple que $\text{Cov}(x, x) = \text{Var}(x)$.

2. Régression linéaire.

Programme une fonction `regression_lineaire(points)` qui à partir d'une liste de points $(x_i, y_i)_{1 \leq i \leq n}$ renvoie les coefficients a, b de la droite d'équation $y = ax + b$.

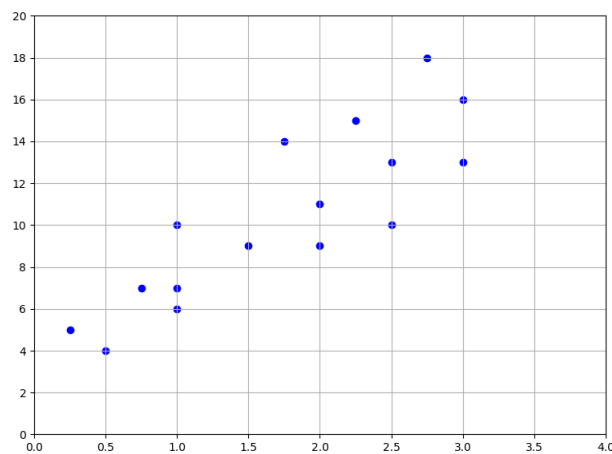
Exemple. Pour le bac blanc on a demandé à chaque élève le temps qu'il a passé pour ses révisions (valeur x_i) et la note qu'il a obtenue (valeur y_i).

Voici la liste (x_i, y_i) des données, par exemple le premier élève a révisé 0.25 heures et a obtenu 5, le dernier à réviser 3 heures et a obtenu 16.

```
eleves = [ (0.25,5), (0.5,4), (0.75,7), (1,6), (1,7),
            (1,10), (1.5,9), (1.75,14), (2,9), (2,11), (2.25,15),
            (2.5,10), (2.5,13), (2.75,18), (3,13), (3,16) ]
```

Calcule les coefficients a et b associés à la régression linéaire.

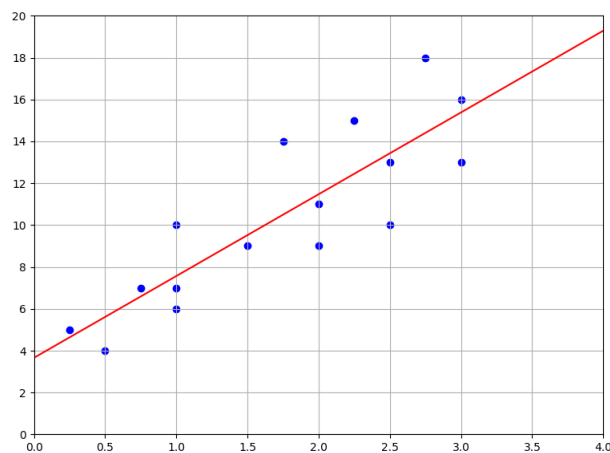
Voici ces points, en abscisse le temps de travail (en heures) et en ordonnée la note obtenue (sur 20) :



J'ai révisé pendant 2 heures, quelle note puis-je espérer ?

3. Affichage.

Programme une fonction `afficher(points)` qui réalise l'affichage des points et de la droite des moindres carrés d'équation $y = ax + b$.



4. Travailler plus pour gagner plus.

Écris un petit programme qui demande à l'utilisateur « Quelle note aimerais-tu avoir ? » et à partir de la réponse donnée affiche une phrase du type « Tu dois travailler au moins 2 heures et 30 minutes. ».

Indications.

- `input()` attend de l'utilisateur une réponse qui est renvoyée sous la forme d'une chaîne de caractères.
- `float(chaine)` transforme une chaîne en un nombre flottant, par exemple `float("12.5")` renvoie 12.5.

Cours 5 (Distribution de Gauss).

Distribution de Gauss.

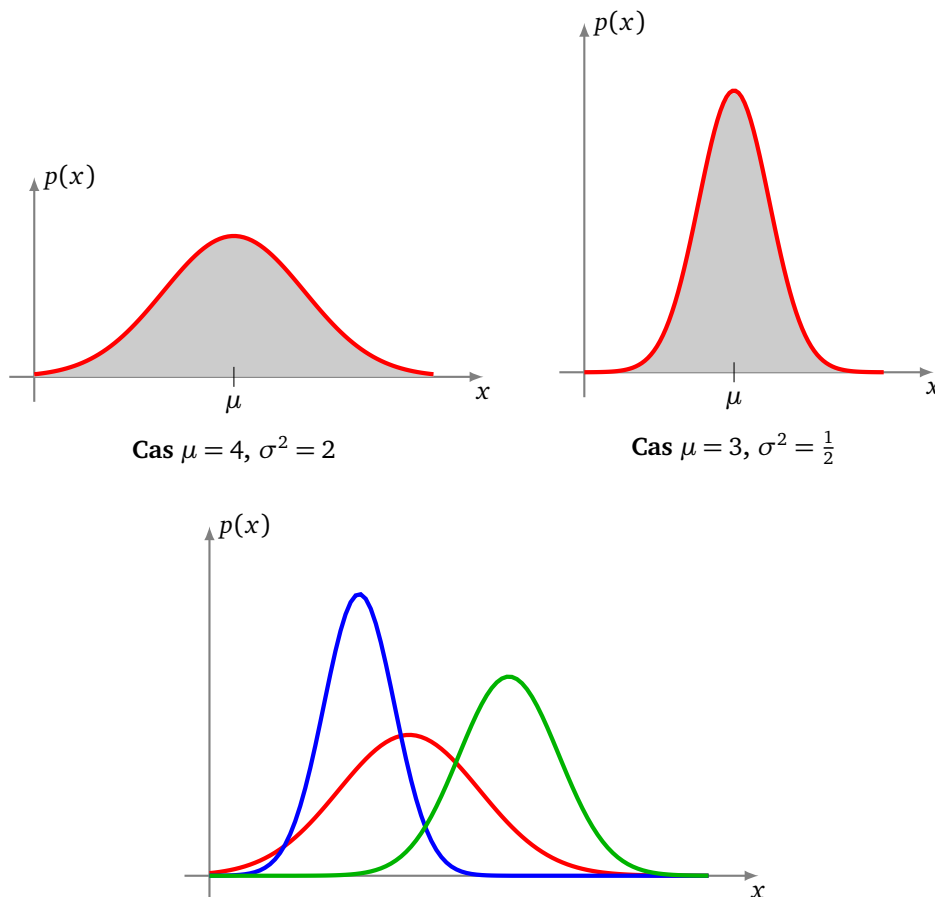
Certaines données se répartissent suivant une **distribution de Gauss** (appelée aussi **loi normale**). Une distribution de Gauss est déterminée par deux paramètres :

- l'espérance (ou la moyenne) μ ,
- la variance (ou l'écart-type au carré) σ^2 ,

et se calcule par la formule :

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2} \frac{(x-\mu)^2}{\sigma^2}\right)$$

Le graphe de cette fonction est une courbe en cloche, centrée sur l'espérance μ , et qui s'étale plus ou moins en fonction de la variance σ^2 .



La fonction $x \mapsto p(x)$ s'appelle une **densité de probabilité**. $p(x)$ n'est pas une probabilité et on peut avoir $p(x) > 1$ pour certains x . Par contre l'aire sous la courbe vaut toujours 1.

Détermination de la distribution à partir d'un échantillon.

Si on nous donne un échantillon de données $x_1, \dots, x_n$ alors on peut lui associer une distribution de Gauss en prenant :

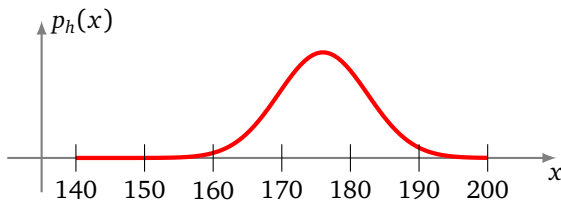
- μ la moyenne de x_i ,

- σ^2 la variance des x_i .

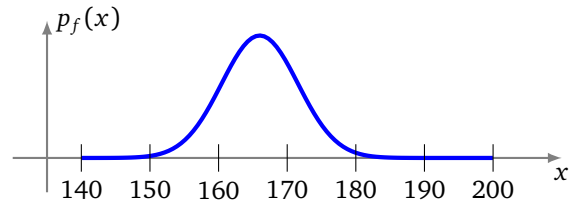
Exemple de la taille. Voici une liste de tailles de 5 hommes : [181, 170, 186, 175, 169]. À partir de cet échantillon on calcule la moyenne $\mu_h = 176$ (arrondie à 1 cm près) et $\sigma_h^2 = 42$. (Tu calculeras les vraies valeurs dans l'activité suivante.) On peut calculer la distribution de Gauss de la taille des hommes (voir le graphique ci-dessous).

On peut faire la même chose à partir de l'échantillon suivant de tailles de femmes : [162, 174, 160, 171, 162]. On trouve $\mu_f = 166$ et $\sigma_f^2 = 32$.

On obtient donc deux densités de probabilité : $p_h(x)$ pour les hommes et $p_f(x)$ pour les femmes.



Distribution de la taille des hommes



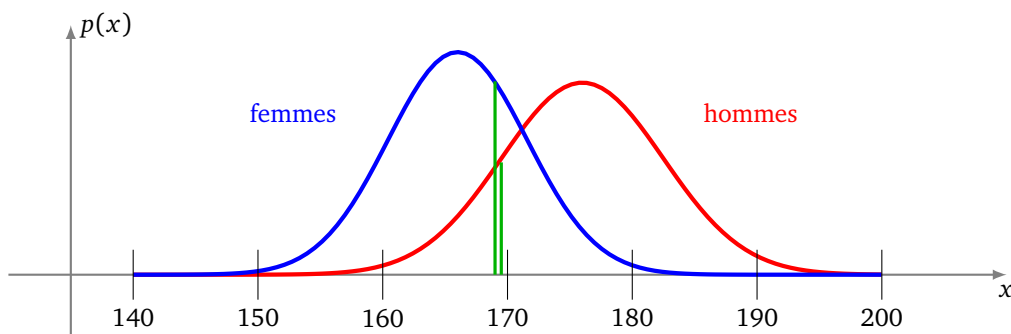
Distribution de la taille des femmes

Classification.

On veut savoir si quelqu'un de taille $x = 169$ cm est plutôt un homme ou une femme ? On a donc deux densités de probabilité : on calcule $p_h(x)$ (comme si c'était un homme) et $p_f(x)$ (comme si c'était une femme) et on compare ces deux valeurs. Si $p_h(x) > p_f(x)$ alors c'est plus probablement un homme, sinon c'est plutôt une femme.

Ici on calcule $p_h(x) \simeq 0.035$ et $p_f(x) \simeq 0.061$, donc avec nos données, quelqu'un de 169 cm est plus probablement une femme.

On peut aussi le voir graphiquement ci-dessous : pour $x = 169$ la courbe des femmes est au-dessus de la courbe des hommes.



Distribution hommes et femmes

Activité 6 (Classification bayésienne naïve).

Objectifs : classer une donnée dans une catégorie ou une autre, par exemple décider si une personne est un homme ou une femme connaissant sa taille et son poids.

1. **Moyenne et variance.** Détermine la moyenne μ_h et la variance σ_h^2 de la taille des hommes à partir de cet échantillon de taille (en cm) :

taille_hommes = [172, 165, 187, 181, 167, 184, 168, 174, 180, 186]

Même chose avec μ_f et σ_f^2 pour les femmes :

`taille_femmes = [172,156,164,182,171,164,162,170,161,167]`

2. **Densité de probabilité.** Programme une fonction `densite_gauss(x, mu, sigma2)` qui renvoie

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2} \frac{(x-\mu)^2}{\sigma^2}\right)$$

où $\mu = \mu$ est une espérance (notre moyenne) et $\text{sigma2} = \sigma^2$ est une variance.

3. **Homme ou femme par la taille.**

On donne la taille x d'une personne, par exemple $x = 170$, on souhaite savoir si cette personne est plus probablement un homme ou une femme. Pour cela calcule la densité de probabilité $p_h(x)$ associée à μ_h et σ_h^2 et la densité de probabilité $p_f(x)$ associée à μ_f et σ_f^2 .

Si $p_h(x) > p_f(x)$ alors il est plus probable que ce soit un homme, sinon c'est plutôt une femme.

Remarques.

- Ce n'est bien sûr pas une certitude ! On va faire mieux dans la question suivante en prenant aussi en compte le poids.
- Les nombres étant très petits, il peut être plus parlant de regarder si $p_h(x)/p_f(x)$ est plus grand ou plus petit que 1.

4. **Par la taille et le poids.**

Voici des échantillons de taille/poids (en cm et kg) pour des hommes puis des femmes.

`hommes = [(172,68), (165,71), (187,85), (181,73), (167,75),
(184,93), (168,67), (174,83), (180,70), (186,73)]`

`femmes = [(172,66), (156,57), (164,48), (182,71), (171,55),
(164,68), (162,52), (170,68), (161,76), (167,67)]`

Pour une taille x , on a maintenant une probabilité $p_h^{\text{taille}}(x)$ et $p_f^{\text{taille}}(x)$. En calculant des moyennes et des variances, on obtient pour un poids y une probabilité $p_h^{\text{poids}}(y)$ et $p_f^{\text{poids}}(y)$. On multiplie les probabilités pour déterminer si une donnée correspond plutôt à un homme ou une femme. On prend une personne de $(\text{taille}, \text{poids}) = (x, y)$. Si on a :

$$p_h^{\text{taille}}(x) \cdot p_h^{\text{poids}}(y) > p_f^{\text{taille}}(x) \cdot p_f^{\text{poids}}(y)$$

alors c'est plus probablement un homme, sinon c'est plus probablement une femme.

Une personne de taille 176 cm pesant 64 kg est-elle plutôt un homme ou une femme ?

Activité 7 (Classification bayésienne naïve (suite)).

Objectifs : classer des phrases dans une catégorie en fonction des mots qu'elle contient.

Voici une liste de titres de sport :

```
titres_sport = [
    "un beau match de championnat",
    "victoire de Paris en finale",
    "défaite à Marseille",
    "le coach viré après la finale",
    "Paris change de coach"]
```

et des titres ne concernant pas le sport :

```
titres_passport = [
    "un beau printemps à Paris",
    "un robot écrase un chien à Marseille",
    "célébration de la victoire de la grande guerre",
    "grève finale au lycée"]
```

À partir de ces titres déjà classés sport/pas sport, tu vas faire déterminer par l'ordinateur si les phrases suivantes parlent de sport :

```
"victoire de Marseille"
"un beau chien"
"Paris écrase Barcelone en finale"
```

1. **Mots.** Programme une fonction `liste_mots(titres)` qui renvoie la liste de tous les mots à partir d'une liste de titres.

Voici le début de la liste obtenue à partir des titres de sports :

```
['un', 'beau', 'match', 'de', 'championnat', 'victoire', 'de', 'Paris', ...]
```

2. **Probabilité d'un mot.** La probabilité qu'un mot m donné soit dans une liste de mots se calcule par la formule :

$$p(m) = \frac{\text{nombre d'occurrences du mot } m}{\text{nombre total de mots}}$$

Par exemple le mot $m = \text{"Paris"}$ est présent 2 fois parmi les 23 mots des titres de sport. Ainsi la probabilité vaut $p(m) = \frac{2}{23} = 0.086\dots$

Programme une fonction `probabilite_mot(mot, liste_mots)` qui renvoie la probabilité du mot donné par rapport à une liste des mots.

3. **Probabilité d'une phrase.**

On définit la probabilité associée à une phrase comme le produit des probabilités des mots qu'elle contient (une liste de mots de titres étant donnée). Si une phrase est formée des mots $m_1, m_2, \dots, m_k$ alors la probabilité de la phrase est :

$$p = p(m_1) \cdot p(m_2) \cdots p(m_k)$$

Par exemple la phrase "la finale de Paris" est formée des mots "la", "finale", "de" et "Paris" qui apparaissent respectivement 1, 2, 3 et 2 fois, pour un total des 23 mots des titres de sport. La probabilité associée à la phrase est donc

$$p = \frac{1}{23} \times \frac{2}{23} \times \frac{3}{23} \times \frac{2}{23} = 0.0000428\dots$$

Programme une fonction `probabilite_phrase(phrase, liste_mots)` qui renvoie la probabilité de la phrase donnée par rapport à une liste de mots donnée.

Application : sport ou pas ?

Pour savoir si la phrase "la finale de Paris" parle de sport ou pas :

- on calcule la probabilité $p(\text{phrase}|\text{sport})$ de la phrase donnée par rapport à la liste des mots des titres parlant de sport,
- on calcule la probabilité $p(\text{phrase}|\text{pas sport})$ de la phrase donnée mais cette fois par rapport à la liste des mots des titres ne parlant pas de sport,
- si $p(\text{phrase}|\text{sport}) > p(\text{phrase}|\text{pas sport})$ alors il est probable que la phrase concerne le sport.

Exemple.

La phrase est "la finale de Paris" :

- on a vu $p(\text{phrase}|\text{sport}) = 0.0000428 \dots$
- on calcule de même :

$$p(\text{phrase}|\text{pas sport}) = \frac{2}{24} \times \frac{1}{24} \times \frac{2}{24} \times \frac{1}{24} = 0.0000120 \dots$$

- Comme $p(\text{phrase}|\text{sport}) > p(\text{phrase}|\text{pas sport})$ alors la phrase concerne probablement du sport (même si les nombres sont petits, l'un est 3 fois plus grand que l'autre).

4. **Probabilité modifiée d'un mot.** On a un problème avec la phrase "le coach perd la finale" car le mot "perd" n'apparaît pas dans nos titres, la probabilité de ce mot est donc $p(m) = 0$. Aussi lorsque l'on calcule la probabilité de la phrase, on obtient $p(\text{phrase}|\text{sport}) = 0$ et $p(\text{phrase}|\text{pas sport}) = 0$ (car on a à chaque fois un facteur nul).

On remédie à ce problème avec une probabilité modifiée et jamais nulle. Elle est définie par :

$$\tilde{p}(m) = \frac{\text{nombre d'occurrences du mot } m + 1}{\text{nombre total de mots}}$$

On a fait comme si n'importe quel mot apparaissait au moins une fois (ce que l'on obtient n'est plus vraiment une probabilité, mais résout notre problème).

Modifie tes fonctions précédentes en :

- `probabilite_mot_bis(mot, liste_mots)` qui renvoie la probabilité modifiée $\tilde{p}$ d'un mot,
- `probabilite_phrase_bis(phrase, liste_mots)` qui renvoie la probabilité de la phrase comme le produit des probabilités modifiées de chaque mot.

Exemple.

La phrase est "le coach perd la finale" :

- $\tilde{p}(\text{phrase}|\text{sport}) = \frac{1+1}{23} \times \frac{2+1}{23} \times \frac{0+1}{23} \times \frac{1+1}{23} \times \frac{2+1}{23} \simeq 5.59 \cdot 10^{-6}$
- $\tilde{p}(\text{phrase}|\text{pas sport}) = \frac{0+1}{24} \times \frac{0+1}{24} \times \frac{0+1}{24} \times \frac{2+1}{24} \times \frac{1+1}{24} \simeq 7.53 \cdot 10^{-7}$
- Comme $\tilde{p}(\text{phrase}|\text{sport}) > \tilde{p}(\text{phrase}|\text{pas sport})$ la phrase concerne très probablement du sport (le premier nombre est 7 fois plus grand que le second).

Conclusion. Les phrases suivantes parlent-elles de sport ou pas ?

"victoire de Marseille"

"un beau chien"

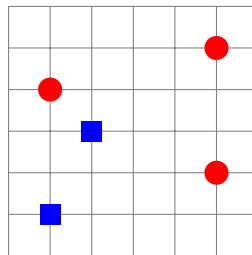
"Paris écrase Barcelone en finale"

C'est aussi avec cette méthode de classification que les courriers électroniques sont filtrés *spam* ou *pas spam*.

L'essor des big-data et de l'intelligence artificielle est dû à l'apparition de nouveaux algorithmes adaptés à la résolution de problèmes complexes : reconnaissance d'images, comportement des électeurs, conduite autonome de voitures... Dans cette seconde partie tu vas programmer quelques algorithmes emblématiques et innovants.

Cours 1 (Les k voisins les plus proches).

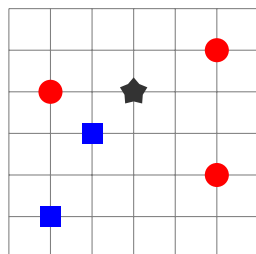
Considérons un électeur qui ne sait pas pour qui voter. Il décide donc de voter comme ses voisins !



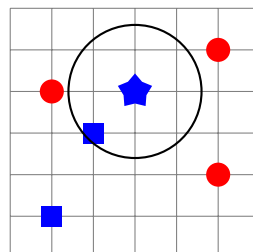
Points de références (rond rouges/carrés bleus)

Pour modéliser la situation, on considère d'abord des points du plan qui peuvent être de deux types, soit des carrés bleus, soit des ronds rouges, on appelle ces points les *points de référence*. Ces points représentent les personnes qui savent déjà pour qui elles vont voter (parmi le choix rouge/bleu). Les autres points n'ont pas encore de couleur, cela représente toutes les personnes qui ne savent pas pour qui elles vont voter.

Voici comment les indécis se décident : pour chaque point non colorié, on regarde le point de référence le plus proche, on colorie alors le point par la couleur de ce point de référence.



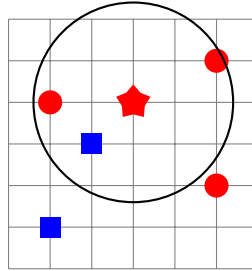
Un indécis (étoile noire)



Coloration de l'indécis par le voisin le plus proche

On généralise cette procédure avec la notion de « k voisins ». Soit k un entier positif. Pour un point non colorié, on cherche les k points de référence les plus proches (ce sont les k voisins). La couleur attribuée est la couleur majoritaire de ces k voisins.

Sur l'exemple ci-dessous, les $k = 3$ voisins sont formés de 2 ronds rouges et 1 carré bleu. On colorie donc l'étoile en rouge.



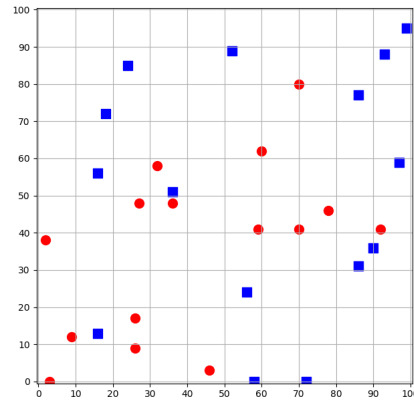
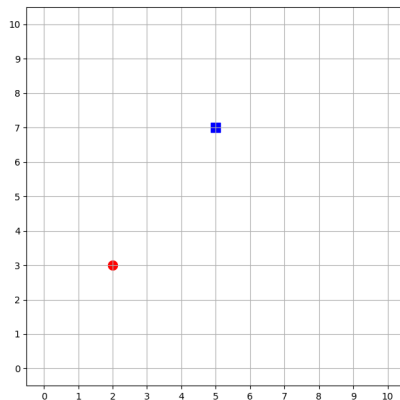
Coloration de l'indécis par
les 3 voisins les plus proches

Le but de l'activité suivante est de colorier tous les points d'une zone grâce à cette méthode.

Activité 1 (Les k voisins les plus proches).

Objectifs : colorier un point en fonction de la couleur de ses voisins.

1. **Préparation.** Programme des fonctions qui permettent d'afficher des points en couleur. On définit un point coloré, appelé *cpoint*, par (x, y, c) où x et y sont des entiers avec $x_{\min} \leq x \leq x_{\max}$ et $y_{\min} \leq y \leq y_{\max}$ et c représente la couleur : $c = 0$ pour du rouge et $c = 1$ pour du bleu.
 - (a) Définis des constantes globales $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$ (par exemple 0, 10, 0, 10) pour la fenêtre des points. Programme une fonction `afficher_cpoints(cpoints)` qui affiche une liste de *cpoints*. Par exemple pour `cpoints = [(2,3,0), (5,7,1)]` cette fonction affiche un point en (2,3) en rouge et un point en (5,7) en bleu (figure de gauche).



- (b) Programme une fonction `fonction_couleur(x,y)` qui renvoie une couleur (0 ou 1) pour un point (x, y) . Tu pourras essayer plusieurs fonctions :
 - 0 ou 1 au hasard,
 - 0 si $((x^2 + y^2) \% 100) - 50 > 0$ et 1 sinon,
 - 0 ou 1 selon le signe de $(x - \frac{x_{\max}}{2})^3 - 3(x - \frac{x_{\max}}{2})(y - \frac{y_{\max}}{2})^2 - x_{\max}$,
 - ou toute autre fonction de ton invention...
- (c) Programme une fonction `generer_cpoints(N)` qui renvoie une liste aléatoire de N *cpoints* (x, y, c) (utilise la fonction précédente pour calculer c). Sur la figure de droite ci-dessus on a affiché 30 points sur $[0, 100] \times [0, 100]$.

2. Le voisin le plus proche.

- (a) Programme une fonction `distance(P,Q)` qui calcule la distance entre $P = (x_1, y_1)$ et $Q = (x_2, y_2)$:

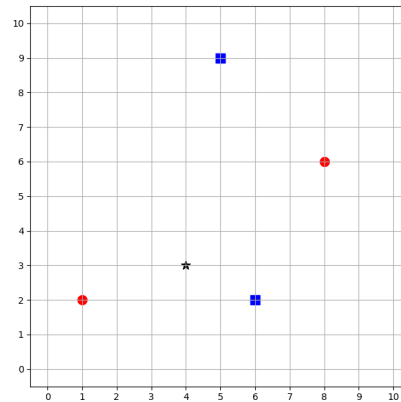
$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

- (b) Programme une fonction `un_voisin_proche(P, cpoints)` qui à partir d'un point $P = (x_0, y_0)$ renvoie un cpoint $Q_c = (x, y, c)$ parmi ceux de la liste donnée et qui est le plus proche possible de P .

Indications.

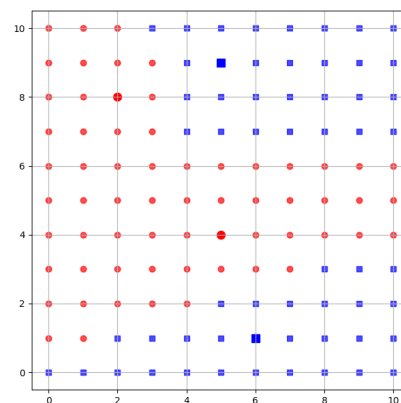
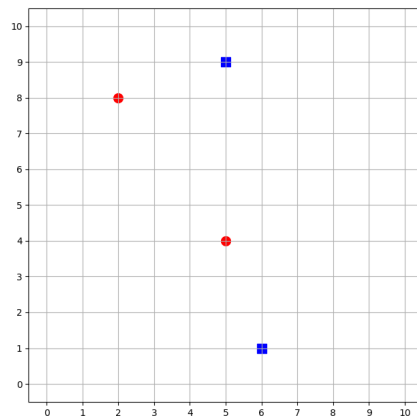
- Il peut y avoir plusieurs voisins à la même distance, la fonction en renvoie un, peu importe lequel.
- Commence par définir un réel $d_{\min}$ très grand (par exemple $d_{\min} = 1000$ ou mieux $d_{\min} = +\infty$ par `dmin = inf` du module `math`). Ensuite calcule la distance entre P et chaque Q_c de la liste et renvoie le plus proche.

Sur l'exemple suivant $P = (4, 3)$ (en noir) a pour plus proche voisin $Q = (6, 2)$ (en bleu) qui est à une distance $d = \sqrt{5}$.

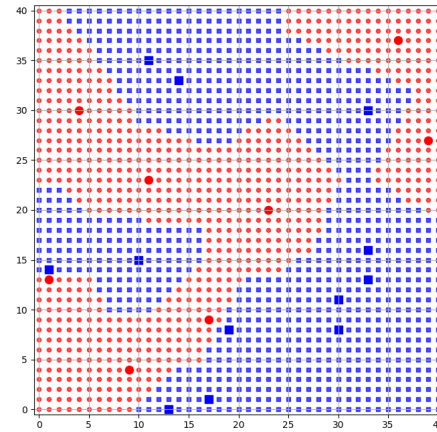
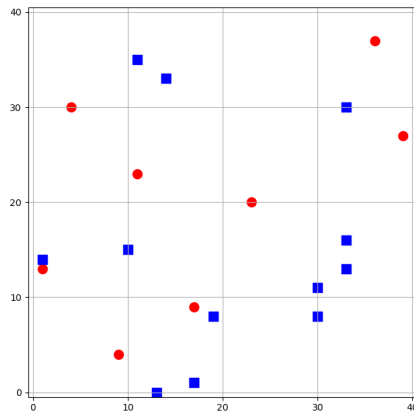


3. **Coloriage.** À partir d'une liste de points colorés, on colorie tous les points : chaque point prend la couleur du voisin le plus proche. Programme ceci en une fonction `colorier_par_un_voisin_proche(cpoints)`.

Ci-dessous à gauche des points colorés initiaux et à droite le coloriage obtenu par la méthode du plus proche voisin.

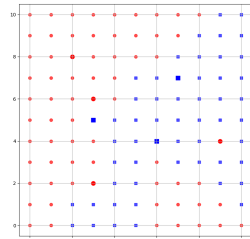
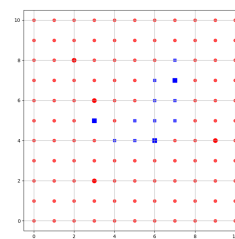
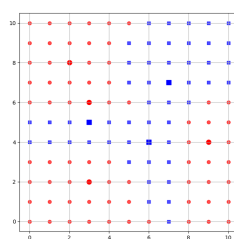
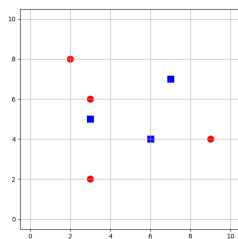


Voici un autre exemple.

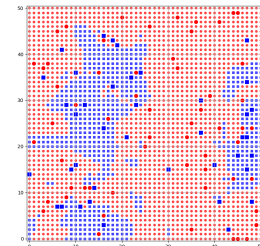
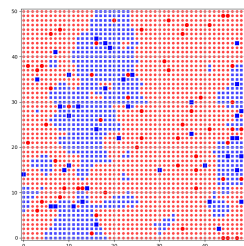
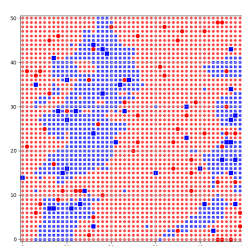
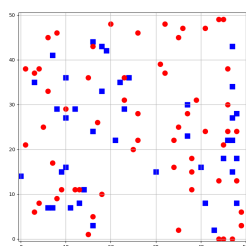


4. Les k voisins proches.

- (a) Programme une fonction `les_voisins_proches(P, cpoints, k)` qui renvoie les k voisins les plus proches du point P .
- $P = (x_0, y_0)$ est un point,
 - `cpoints` est la liste des cpoints (x, y, c) initiaux,
 - $k \geq 1$ est un entier,
 - la fonction renvoie une liste de k cpoints.
- (b) Programme une fonction `couleur_majoritaire(cpoints)` qui renvoie la couleur majoritaire (0 ou 1) d'une liste de cpoints.
- (c) Programme une fonction `colorier_par_les_voisins_proches(cpoints, k)` qui à partir d'une liste de points colorés initiaux, colorie tous les autres points : chaque point prend la couleur majoritaire des k voisins les plus proches ($k = 1$ correspond au voisin le plus proche).
- Ci-dessous à gauche les points colorés initiaux et à droite des coloriages pour différentes valeurs de k ($k = 1, 2, 3$).



Et voici un exemple de taille plus grande avec $k = 3, 5, 7$.



Cours 2 (Expression correctement parenthésée).

Voici des exemples d'expressions bien et mal parenthésées :

- $2 + (3 + b) \times (5 + (a - 4))$ est correctement parenthésée ;
- $(a + 8) \times 3) + 4$ est mal parenthésée : il y a une parenthèse fermante «) » seule ;
- $(b + 8/5)) + (4$ est mal parenthésée : il y a autant de parenthèses ouvrantes « (» que de parenthèses fermantes «) » mais elles sont mal positionnées.

Dans le chapitre « Calculatrice polonaise – Piles » du premier tome, nous avons étudié un algorithme qui vérifie si une expression a ses parenthèses bien agencées. Cette méthode résout complètement le problème mais nécessite beaucoup de mémoire. En effet la taille de la pile peut être de l'ordre de grandeur de la longueur de l'expression, ce qui pose des problèmes pour des expressions ayant des millions de caractères. Par contre, avec la méthode de l'activité suivante tu ne stockes que les valeurs de deux entiers (h et S), l'inconvénient c'est que la réponse renvoyée est probablement vraie, mais ce n'est pas une certitude. Cette méthode permet aussi de vérifier si un fichier (par exemple un fichier « xml ») a un balisage correct.

Activité 2 (Parenthèses correctes?).

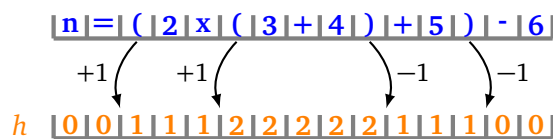
Objectifs : tester si une expression a ses parenthèses et ses crochets bien agencés.

Partie A. Parenthèses seules.

On considère une expression avec des parenthèses (tous les autres caractères sont ignorés). On associe une hauteur h en lisant l'expression de gauche à droite :

- au départ $h = 0$,
- avant une parenthèse ouvrante "(" on augmente h de 1,
- après une parenthèse fermante ")" on diminue h de 1.

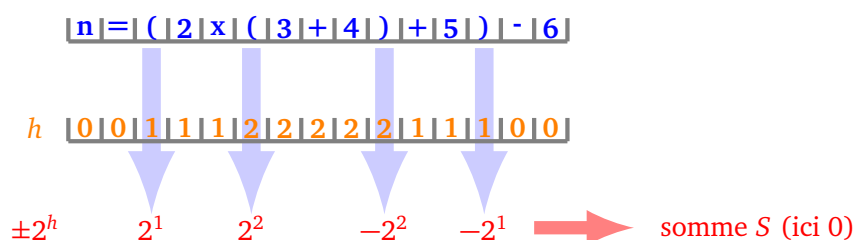
Voici un exemple avec l'expression « $n = (2 \times (3 + 4) + 5) - 6$ ».



On fixe un nombre premier p (par exemple $p = 11$ ou $p = 101$) et on pose $a = 2$. À chaque parenthèse on associe un entier modulo p :

$$" (" \mapsto 2^h \pmod{p} \quad \text{et} \quad ")" \mapsto -2^h \pmod{p},$$

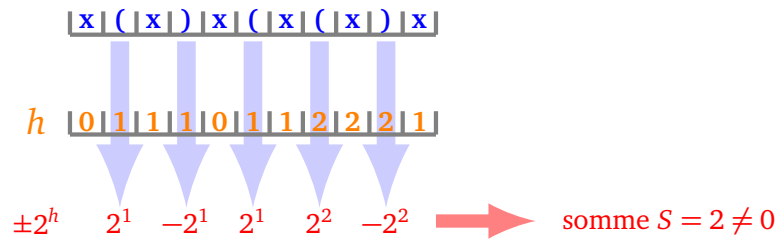
à tous les autres caractères on associe 0. Ensuite on calcule la somme S de tous ces termes.



- Si l'expression est bien parenthésée alors $S = 0$.
- Si à un moment $h < 0$ ou si à la fin $S \neq 0$ alors l'expression est mal parenthésée.

Programme cette méthode en une fonction `test_parentheses(expression)` qui renvoie « Vrai » si la somme de contrôle vaut 0 et « Faux » sinon.

Voici un exemple d'une expression mal parenthésée : « $x(x)x(x)x$ » (ici x désigne n'importe quelle suite de caractères autre que des parenthèses), le problème est bien détecté car la somme $S = 2$ n'est pas nulle.



Partie B. Parenthèses et crochets.

On va améliorer cette fonction pour tester une expression avec des parenthèses et des crochets. Voici une expression cohérente « $[(a + b) \times (a - b)]$ » ; voici des expressions non correctes « $[a + b]$ » ou « $(a \times [b + c] - d)$ ».

On considère une expression avec des parenthèses et des crochets (tous les autres caractères sont ignorés). On associe une hauteur h en lisant l'expression de gauche à droite, au départ $h = 0$, on augmente h de 1 avant une parenthèse ou un crochet ouvrant, "(" ou "[", on diminue h de 1 après une parenthèse ou un crochet fermant, ")" ou "]" .

On fixe un nombre premier p assez grand (par exemple $p = 11$ ou $p = 101$) et on pose $a = 2$ et $b = 3$. À chaque parenthèse on associe comme auparavant un entier modulo p :

$$" (" \mapsto 2^h \pmod{p} \quad \text{et} \quad ")" \mapsto -2^h \pmod{p}.$$

À chaque crochet on associe aussi un entier modulo p :

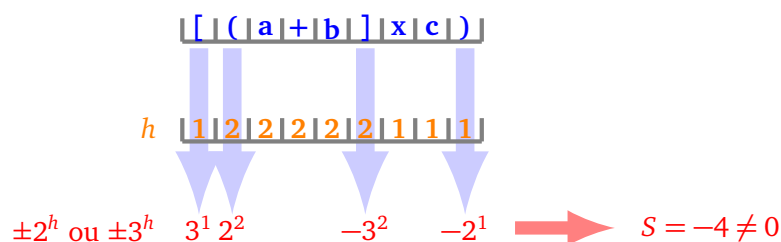
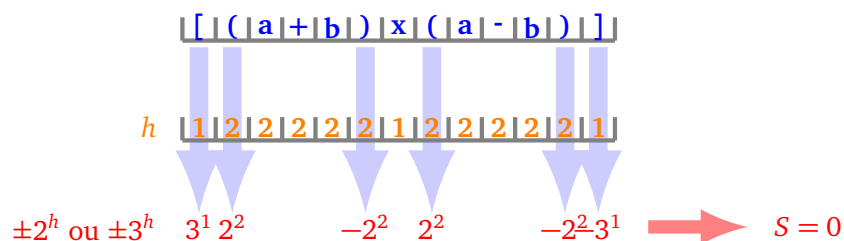
$$" [" \mapsto 3^h \pmod{p} \quad \text{et} \quad "]" \mapsto -3^h \pmod{p}.$$

À tous les autres caractères on associe 0. Ensuite on calcule la somme S de tous ces termes.

- Si l'expression est bien parenthésée et crochétée alors $S = 0$.
- Si à un moment $h < 0$ ou si à la fin $S \neq 0$ alors l'expression est mal parenthésée ou mal crochétée.
- Il peut y avoir des cas exceptionnels où $S = 0$ et cependant l'expression est mal parenthésée/crochétée.

Programme une fonction `test_crochets_parentheses(expression)` qui renvoie « Vrai » si la somme de contrôle vaut 0 et « Faux » sinon.

Voici deux exemples.



Teste ta fonction sur les deux exemples suivants :

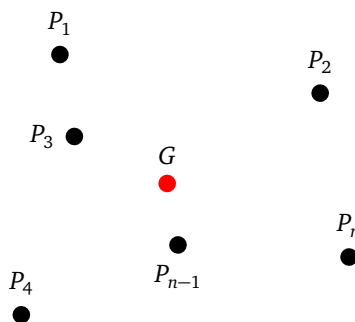
```
"([[]][[[[()([()([(([]))]]))]]])"
"[[[()](()([[[[]])()([[[[]]([()]]))]])]"
```

On a une certitude uniquement dans le cas où $S \neq 0$ (l'expression est alors mal parenthésée/crochetée), dans le cas $S = 0$ on a seulement une forte probabilité que l'expression soit bien parenthésée/crochetée. Voici des exemples de situations non détectées. Par exemple " $[]$ ($[]$ " (mais si on regarde juste les parenthèses on voit que l'expression n'est déjà pas correcte). Il faut donc au préalable vérifier que les parenthèses seules sont bien positionnées, et aussi que les crochets seuls sont bien positionnés. Autre soucis possible avec " $($ " et $p = 11$ alors on trouve une somme $S = 2^1 + 3^2 = 11$ donc $S = 0 \pmod{11}$ et pourtant l'expression est mal parenthésée/crochetée. Par contre avec $p = 7$, on trouve $S = 4 \pmod{7}$, donc cela prouve que l'expression est mal parenthésée/crochetée.

Note. Cet algorithme peut être programmé en temps réel, c'est-à-dire que l'on peut commencer les calculs dès la lecture du premier caractère et les terminer juste après le dernier.

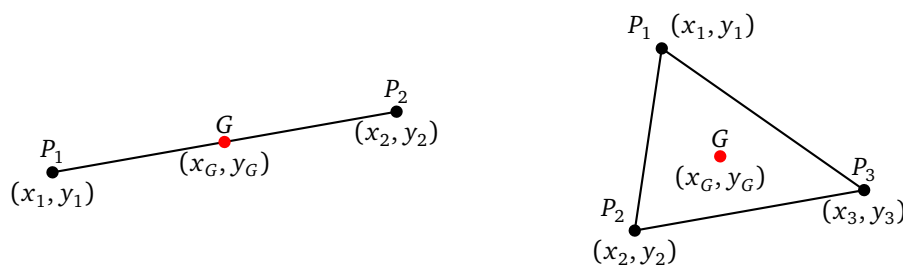
Cours 3 (Isobarycentre).

On considère n points du plan $P_1, P_2, \dots, P_n$. On cherche un point « le plus au milieu possible » de tous ces points. On appelle ce point **l'isobarycentre** des $\{P_i\}$ et on le note G .



Cas de deux points. Si $n = 2$ alors G est simplement le milieu du segment $[P_1, P_2]$.

Cas de trois points. Si $n = 3$ alors G est le centre de gravité du triangle défini par les trois points. C'est donc l'intersection des médianes.



Formule. Pour n quelconque, voici la formule pour calculer les coordonnées (x_G, y_G) de l'isobarycentre G en fonction des coordonnées (x_i, y_i) des P_i ($i = 1, \dots, n$) :

$$x_G = \frac{x_1 + x_2 + \dots + x_n}{n} \quad \text{et} \quad y_G = \frac{y_1 + y_2 + \dots + y_n}{n}$$

Cela signifie que l'abscisse de G est simplement la moyenne des abscisses des P_i et l'ordonnée de G est la moyenne des ordonnées des P_i . Pour $n = 2$ on retrouve les coordonnées du milieu $(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2})$. Et pour $n = 3$ cela donne

$$(x_G, y_G) = \left(\frac{x_1 + x_2 + x_3}{3}, \frac{y_1 + y_2 + y_3}{3} \right).$$



Activité 3 (Barycentres).

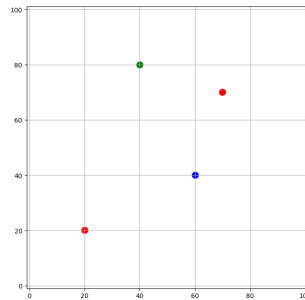
Objectifs : rassembler des points en groupes.

Note. Certaines fonctions sont très proches de la première activité sur les k voisins.

1. Afficher des points.

Programme d'abord une fonction `afficher_points(points, couleurs)` qui affiche une liste de points (x, y) en fonction d'une liste de couleurs (une couleur par point).

Par exemple avec `points = [(20,20), (60,40), (40,80), (70,70)]` et `couleurs = [0,1,2,0]` alors `afficher_points(points, couleurs)` affiche le graphique ci-dessous.

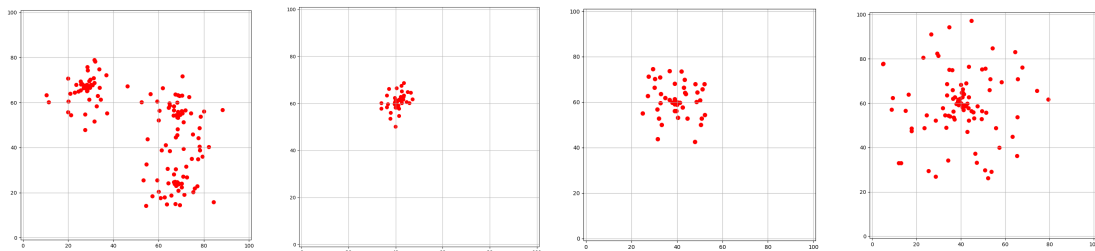


Indications.

- Tu peux définir des constantes $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$ pour définir une fenêtre de visualisation. On prendra par défaut $[0, 100] \times [0, 100]$.
- Pour les couleurs, 0 peut être pour le rouge, 1 pour le bleu...

2. Générer des points.

On veut générer des groupes aléatoires de points, comme ci-dessous à gauche avec trois groupes de points. Chaque groupe est généré au hasard autour d'un centre, avec un paramètre de dispersion qui fait que les points s'éloignent plus ou moins du centre (voir les trois figures de droite : un seul groupe de points mais avec des paramètres de dispersion différents).



Voici comment programmer une fonction `generer_points(k, n, d)` qui affiche k groupes, de n points chacun, selon une dispersion d .

Répéter k fois :

- Choisir un centre (x_c, y_c) au hasard dans la fenêtre (c'est mieux s'il n'est pas trop près des bords).
- Pour ce centre, répéter n fois :
 - Choisir un angle θ au hasard entre 0 et 2π .
 - Choisir un rayon r au hasard selon la formule $r = d\rho^2$ où ρ est un réel tiré au hasard entre 0 et 1.
 - Ajoute le point (x, y) à la liste des points, où :

$$x = x_c + r \cos \theta \quad \text{et} \quad y = y_c + r \sin \theta$$

(vérifie que (x, y) est bien dans la fenêtre voulue).

Indications. La fonction `random()` (sans argument) du module `random` renvoie à chaque appel un nombre flottant aléatoire x de $[0, 1[$. Pour obtenir à partir de x un nombre aléatoire y entre a et b , on peut utiliser la formule :

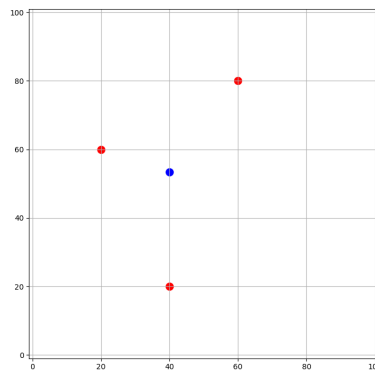
$$y = a + (b - a)x.$$

3. Calcul du barycentre.

Programme une fonction `calcul_barycentre(points)` qui calcule l'isobarycentre (x_G, y_G) d'une liste de points $\{(x_i, y_i)\}$. On rappelle la formule :

$$x_G = \frac{x_1 + x_2 + \dots + x_n}{n} \quad \text{et} \quad y_G = \frac{y_1 + y_2 + \dots + y_n}{n}.$$

Exemple. Avec les points $(20, 60)$, $(40, 20)$, $(60, 80)$ (en rouge) la fonction calcule les coordonnées du barycentre $(x_G, y_G) = (40, 53.33\dots)$ (en bleu).



4. Barycentre le plus proche.

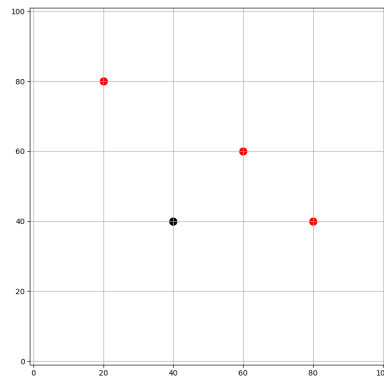
Dans cette question on se donne une liste de barycentres (on verra plus tard comment on les obtient) et une liste de points. Pour chaque point on va trouver quel est le barycentre le plus proche.

- (a) Programme une fonction `distance(P, Q)` qui calcule la distance entre $P = (x_1, y_1)$ et $Q = (x_2, y_2)$:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

- (b) Programme une fonction `barycentre_proche(P, barycentres)` qui à partir d'un point $P = (x_0, y_0)$ renvoie le rang du point le plus proche dans la liste `barycentres` donnée.

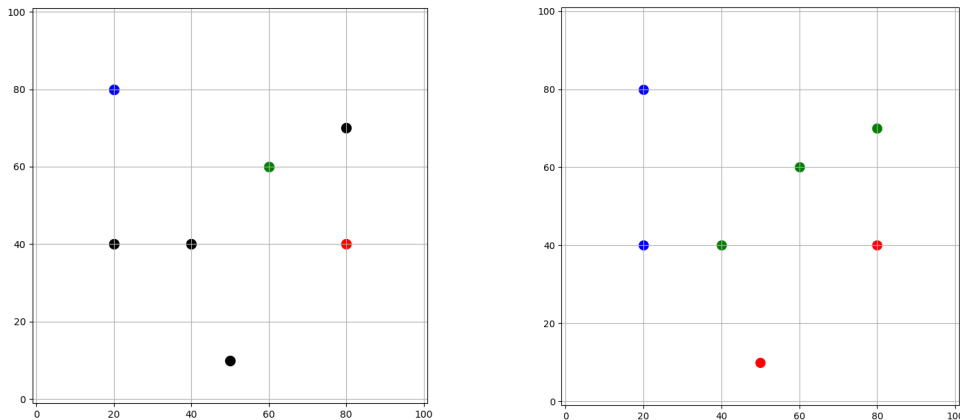
Par exemple, si nous avons la liste `barycentres = [(80, 40), (20, 80), (60, 60)]` (en rouge) et le point $P = (40, 40)$ (en noir), alors le barycentre le plus proche de P est $G_2 = (60, 60)$ qui est de rang 2 dans la liste des barycentres. Donc la fonction `barycentre_proche(P, barycentres)` renvoie ici 2.



(c) **Coloriage suivant le barycentre le plus proche.** La fonction précédente renvoie le rang du barycentre le plus proche et on considère ce rang comme la couleur (le barycentre G_0 en tête de liste est de couleur 0 donc rouge, le barycentre G_1 est de couleur 1 donc bleu...).

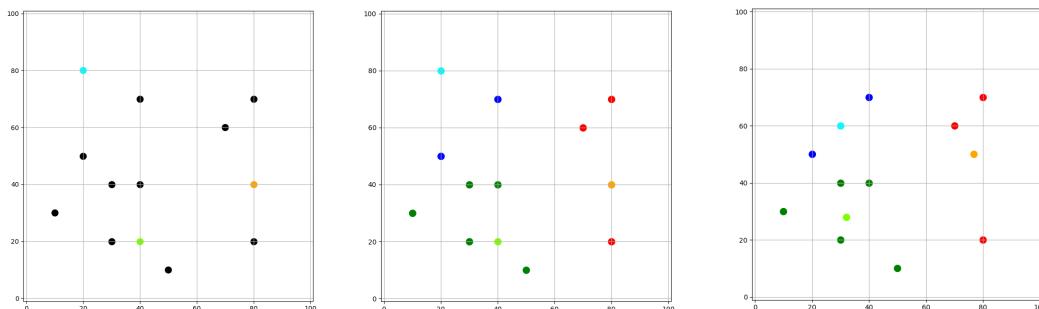
Déduis-en une fonction `couleurs_barycentres_proches(points, barycentres)` qui à partir d'une liste de points renvoie la liste des couleurs attribuées à chaque point (selon le rang du barycentre le plus proche, c'est-à-dire la couleur).

Exemple. Reprenons nos barycentres `barycentres = [(80,40), (20,80), (60,60)]` avec chacun une couleur donnée par son rang : $G_0 = (80,40)$ est de couleur 0 (rouge), $G_1 = (20,80)$ est de couleur 1 (bleu), $G_2 = (60,60)$ est de couleur 2 (vert). Fixons aussi une liste de points `points = [(40,40), (20,40), (80,70), (50,10)]` pour l'instant sans couleur (en noir sur la figure de gauche ci-dessous). Alors la fonction `couleurs_barycentres_proches()` renvoie la liste `[2, 1, 2, 0]`. Cela signifie que le point $P_0 = (40,40)$ doit être colorié par la couleur 2 car le plus proche barycentre est G_2 , le point $P_1 = (20,40)$ doit être colorié par la couleur 1 car le plus proche barycentre est G_1 ... Cela permet de colorier les quatre points (figure de droite).



5. Recalculer les barycentres.

- On se donne une liste de points et une liste de barycentres (figure de gauche ci-dessous, pour l'instant cette liste de « barycentres » est arbitraire).
- On a vu comment attribuer une couleur à chaque point (figure centrale), ainsi les points sont regroupés par couleur. Les points bleus sont associés au barycentre bleu clair, les points verts sont associés au barycentre vert clair...
- Pour tous les points d'une même couleur, on calcule le barycentre. On se retrouve avec une nouvelle liste de barycentres (figure de droite) (cette fois ce sont de « vrais » barycentres).



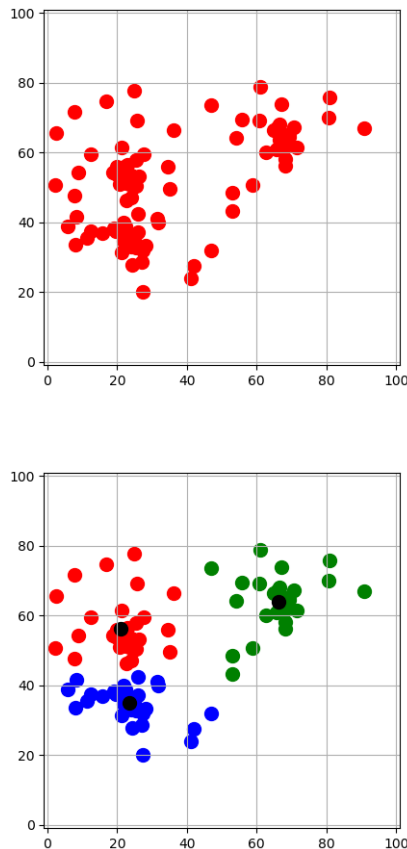
Programme une fonction `recalculer_barycentres(points, barycentres)` qui effectue cette tâche. Sur l'exemple graphique ci-dessus les barycentres de départ sont `[(80,40), (20,80), (40,20)]`

(figure de gauche), la fonction `recalculer_barycentres()` renvoie les coordonnées des nouveaux barycentres : $[(76.66\dots, 50), (30, 60), (32, 28)]$.

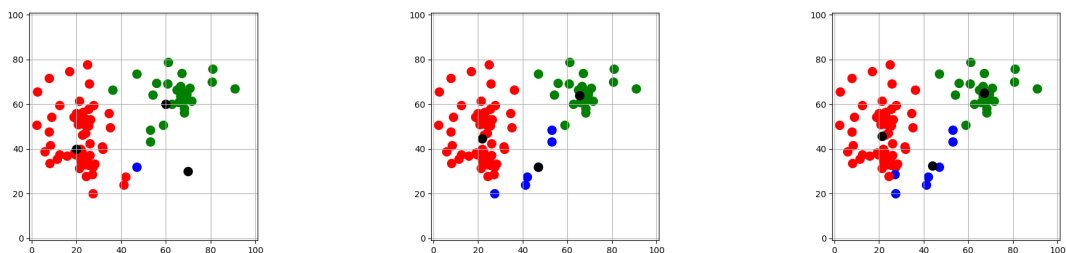
Indications. Si un barycentre n'est associé à aucun point on le conserve inchangé.

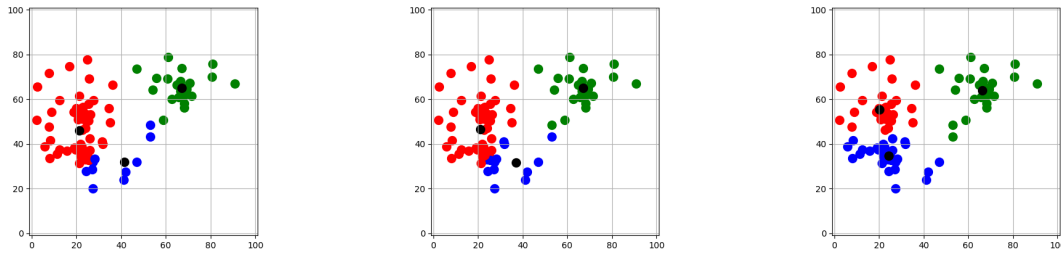
6. Itérer les barycentres.

Voici comment regrouper des points en les coloriant par groupe. Sur la figure de gauche on a tiré des points (presque) au hasard (on devine à peine 3 groupes). Sur la figure de droite, l'algorithme a séparé les points en 3 groupes pertinents (avec en noir les 3 barycentres que l'on va déterminer).



Voici les étapes graphiques de l'algorithme. Sur la figure de gauche ci-dessous, on part de trois points (en noir) qui jouent le rôle de barycentres initiaux (on les choisit au hasard ou bien trois points assez écartés de la fenêtre). On colorie les autres points selon le barycentre le plus proche. Étape suivante (figure du milieu) on calcule le nouveau barycentre pour les points rouges, le nouveau barycentre pour les points bleus... mais alors on doit aussi recalculer la couleur de tous les autres points (puisque les coordonnées des barycentres ont changé). On itère le processus (autres figures) jusqu'à ce que les couleurs des points ne changent plus.



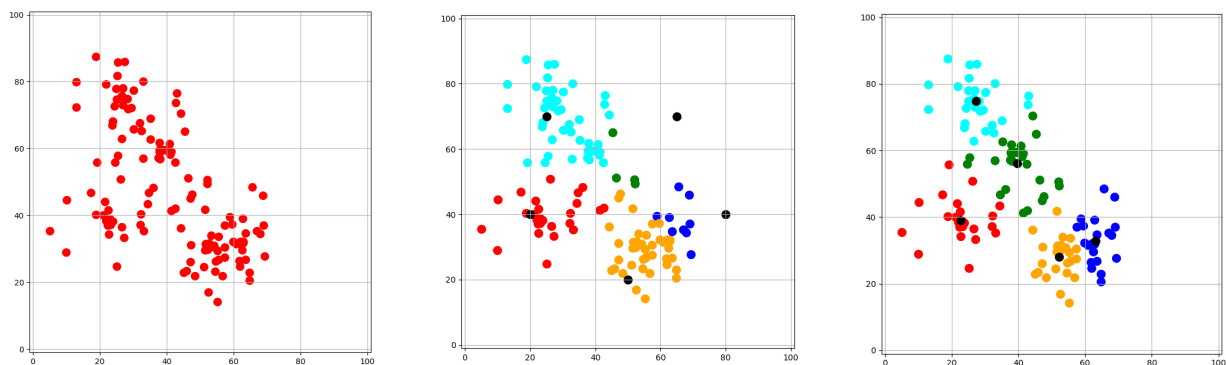


Programme une fonction `iterer_barycentres(points, barycentres_init)` qui effectue cette tâche.

Indications.

- Il s'agit juste, à chaque étape, d'utiliser la fonction `recalculer_barycentres()` suivie de `couleurs_barycentres_proches()` avec les nouvelles coordonnées des barycentres.
- On itère ces étapes jusqu'à ce que la liste des couleurs se stabilise (ou bien après un nombre fixé d'étapes).
- Pour remplacer la liste des anciens barycentres par la liste des nouveaux, utilise :
`barycentres = list(nouv_barycentres)`

On termine par un exemple avec 5 groupes de points (à gauche les points de départ, au centre les 5 barycentres initiaux et le premier coloriage, à droite le résultat après plusieurs itérations). Il faut parfois tester différentes configurations des barycentres initiaux pour que cela fonctionne bien.



Note.

Il y a une différence fondamentale entre cette activité et l'activité sur les k voisins. Il s'agit ici d'un *apprentissage non supervisé*, c'est-à-dire qu'on laisse l'algorithme trouver tout seul les groupes. L'activité sur les k voisins est celle d'un *apprentissage supervisé* : certains points sont déjà coloriés rouge ou bleu et servent de référence, ensuite il s'agit de colorier les autres points suivant ce modèle.

Cours 4 (Neurone).

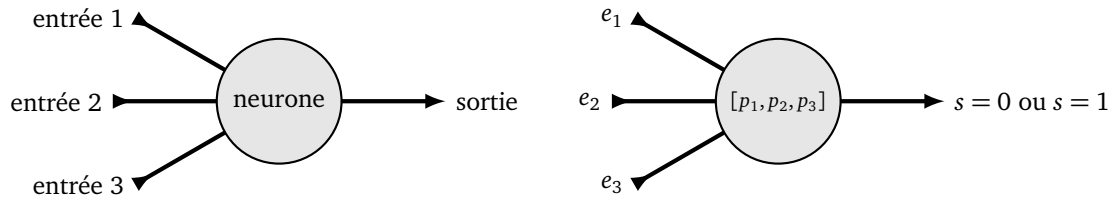
Des neurones.

Le cerveau est composé d'environ 100 milliards de neurones. Les réseaux de neurones artificiels s'inspirent du cerveau pour répondre à des problèmes complexes. Un réseau de neurones possède des coefficients qui sont calculés par apprentissage. Le réseau est ensuite capable de répondre au problème posé, y compris lors de situations qui n'ont pas été rencontrées lors de l'apprentissage.

Un neurone.

Nous allons travailler avec un seul neurone. Notre modèle du neurone s'appelle le *perceptron*. Notre neurone fonctionne ainsi :

- Il reçoit en entrée des nombres, pour nous ce sera 3 réels e_1, e_2, e_3 .
- En sortie il renvoie un entier s qui vaut 0 ou 1. Si la sortie vaut 1, on dit que le neurone est *activé*.
- Chaque neurone est déterminé par des nombres, pour nous 3 réels p_1, p_2, p_3 . C'est *l'état* du neurone.



Activation du neurone.

Comment savoir si le neurone est activé? En fonction de l'entrée (e_1, e_2, e_3) et de l'état du neurone (p_1, p_2, p_3) , on commence par calculer une valeur q selon la formule :

$$q = p_1 e_1 + p_2 e_2 + p_3 e_3.$$

Ensuite on regarde si q est plus grand ou plus petit que 1 pour déterminer si le neurone est activé (c'est-à-dire la valeur s renvoyée par le neurone) :

$$\begin{cases} s = 0 & \text{si } q < 1, \\ s = 1 & \text{si } q \geq 1. \end{cases}$$

Note bien qu'il n'y a que deux sorties possibles $s = 0$ ou $s = 1$.

Objectif.

Quelle tâche demande-t-on à notre neurone? Le neurone prend en entrée des nombres (e_1, e_2, e_3) et renvoie en sortie 0 ou 1. Il répond donc à une question du type « oui ou non? ». Plus précisément la question à laquelle répond le neurone est « ce point de coordonnées (e_1, e_2, e_3) est-il au-dessus ou en dessous du plan $\mathcal{P}$? ». Le plan $\mathcal{P}$ dont il est question est le plan d'équation $p_1 x + p_2 y + p_3 z = 1$ déterminé par l'état (p_1, p_2, p_3) du neurone. On verra dans l'activité suivante comment cela permet de répondre à la question « cette couleur est-elle rouge ou pas? ».

Le problème principal est, qu'au départ, on ne connaît pas le plan $\mathcal{P}$ qui répond au problème que l'on se pose, autrement dit on ne sait pas quel état (p_1, p_2, p_3) convient. Pour notre exemple cela signifie que l'ordinateur ne sait pas ce qu'est la couleur rouge. Il va donc falloir lui apprendre !

Apprentissage.

On part d'un état initial quelconque par exemple $(p_1, p_2, p_3) = (1, 1, 1)$. Pour faire évoluer l'état du neurone jusqu'à la bonne valeur de (p_1, p_2, p_3) on va l'entraîner comme un enfant : on lui montre une couleur et on lui dit « c'est du rouge », puis on lui montre une autre couleur et on lui dit « ce n'est pas du rouge ». À chaque étape l'état du neurone (p_1, p_2, p_3) est modifié.

Voici une étape d'entraînement : on donne une entrée (e_1, e_2, e_3) et l'objectif $b = 0$ ou $b = 1$ qui est la sortie attendue si le neurone était parfaitement paramétré. On calcule la sortie $s = 0$ ou $s = 1$ que renvoie le neurone selon l'entrée (e_1, e_2, e_3) dans son état actuel, ensuite plusieurs cas sont possibles :

- Si l'objectif b est égal à la sortie s alors le neurone fonctionne bien pour cette entrée, on ne change pas l'état du neurone.
- Si la sortie calculée est $s = 0$ alors que l'objectif est $b = 1$, alors on change l'état du neurone (p_1, p_2, p_3)

en un nouvel état (p'_1, p'_2, p'_3) selon la formule :

$$\begin{cases} p'_1 = p_1 + \epsilon e_1 \\ p'_2 = p_2 + \epsilon e_2 \\ p'_3 = p_3 + \epsilon e_3 \end{cases}$$

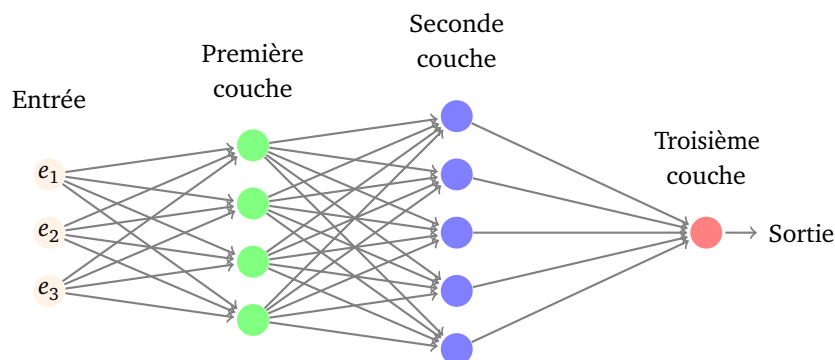
- Si la sortie calculée est $s = 1$ alors que l'objectif est $b = 0$, alors on change l'état du neurone selon la formule :

$$\begin{cases} p'_1 = p_1 - \epsilon e_1 \\ p'_2 = p_2 - \epsilon e_2 \\ p'_3 = p_3 - \epsilon e_3 \end{cases}$$

Le paramètre ϵ est un petit réel. On prendra $\epsilon = 0.2$ par exemple. On répète ces étapes avec plusieurs couples (entrées, objectifs). Géométriquement chaque apprentissage déplace un petit peu le plan $\mathcal{P}$ pour mieux répondre au problème. L'état du neurone va converger vers un état (p_1, p_2, p_3) . Une fois la phase d'apprentissage terminée, on conserve cet état final (p_1, p_2, p_3) . Maintenant on laisse répondre le neurone en regardant s'il s'active ou pas selon des entrées quelconques (e_1, e_2, e_3) (même si ces entrées ne font pas partie de la liste d'apprentissage).

Réseau de neurones.

Dans un réseau de neurones il y a plusieurs neurones organisés en couches. Les sorties d'une couche servent d'entrées pour la couche suivante. Un réseau de neurones simple et bien entraîné peut reconnaître des chiffres manuscrits (c'est un 0, c'est un 1...), des réseaux plus sophistiqués reconnaissent des photos (c'est un chat ou c'est un chien), jouent aux échecs (le meilleur coup est la reine en d7) ou conduisent des voitures !



Cours 5 (Couleurs).

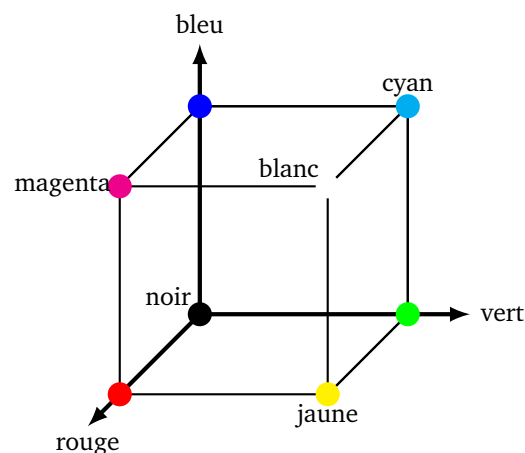
Code couleur *rgb*. Dans le système *rgb*, on code une couleur selon ses niveaux de rouge/vert/bleu (*red/green/blue*). Une couleur est donc codée par trois réels (r, g, b) chacun entre de 0 à 1.

Voici quelques exemples de couleurs.

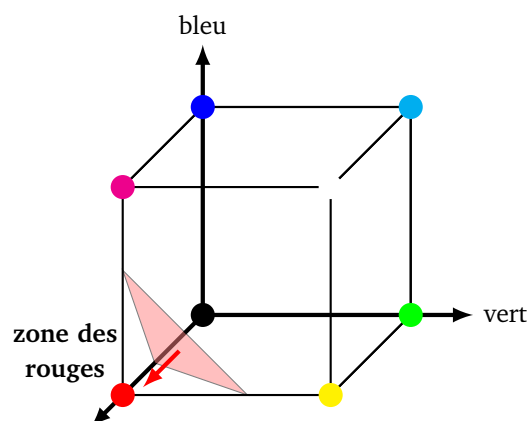
Couleur	Nom	Niveau de rouge	Niveau de vert	Niveau de bleu
	rouge	1	0	0
	vert	0	1	0
	bleu	0	0	1
	blanc	1	1	1
	noir	0	0	0
	orange	1	0.5	0
	gris	0.5	0.5	0.5
	cyan	1	1	0
	une nuance de rouge	0.8	0.2	0.1

Nuances de rouge. Dans l'usage, la couleur rouge ne se limite pas au code (1, 0, 0), il peut y avoir du rouge clair, du rouge foncé, du rouge orangé... Par exemple le code (0.8, 0.2, 0.1) est bien une nuance de rouge (voir le tableau ci-dessus).

Si on considère l'espace de toutes les couleurs comme un cube (axe x pour le rouge variant de 0 à 1, axe y pour le vert, axe z pour le bleu) alors le rouge et ses nuances correspondent à une zone au voisinage du point (1, 0, 0).



Dans l'activité suivante nous allons programmer un neurone afin qu'il détecte une zone de rouge. Cette zone sera une portion du cube des couleurs coupé par un plan d'équation $p_1x + p_2y + p_3z = 1$ (où (x, y, z) jouent le rôle de (r, g, b)). Par apprentissage, on va déterminer les coefficients (p_1, p_2, p_3) qui conviennent.





Activité 4 (Neurone).

Objectifs : programmer un neurone qui après un apprentissage détecte si une couleur donnée est rouge ou pas.

1. Activation d'un neurone.

Programme une fonction `activation(neurone,entree)` qui selon l'état du neurone = $[p_1, p_2, p_3]$ et les valeurs `entree` = $[e_1, e_2, e_3]$ renvoie $s = 1$ en cas d'activation et $s = 0$ sinon. On rappelle que cette activation est déterminée par :

$$\begin{cases} s = 1 & \text{si } p_1 e_1 + p_2 e_2 + p_3 e_3 \geq 1, \\ s = 0 & \text{sinon.} \end{cases}$$

Exemples.

- Avec `neurone` = $[1, 2, 3]$ et `entree` = $[0.5, 0, 0]$ alors on calcule $q = p_1 e_1 + p_2 e_2 + p_3 e_3 = 1 \times 0.5 + 2 \times 0 + 3 \times 0 = 0.5 < 1$. Donc le neurone ne s'active pas, la sortie vaut $s = 0$.
- Avec `neurone` = $[1, 2, 3]$ et `entree` = $[0, 1, 0.5]$ alors on calcule $q = p_1 e_1 + p_2 e_2 + p_3 e_3 = 1 \times 0 + 2 \times 1 + 3 \times 0.5 = 3.5 \geq 1$. Cette fois le neurone s'active et la sortie vaut $s = 1$.
- Pour `neurone` = $[1, 0.5, 2]$ et `entree` = $[0.2, 0.1, 0.1]$ est-ce que le neurone s'active ou pas ? Et avec `entree` = $[0.3, 0.2, 0.7]$?

2. Apprentissage.

Programme une fonction `apprentissage(neurone,entree,objectif)` qui renvoie l'état $[p'_1, p'_2, p'_3]$ modifié du neurone après apprentissage avec l'entrée et l'objectif (0 ou 1) donnés.

Voici comment faire :

- On note l'état actuel du neurone par $[p_1, p_2, p_3]$ et l'entrée donnée par $[e_1, e_2, e_3]$.
- On commence par calculer si le neurone s'active ou pas avec cette entrée. On note $s = 0$ ou $s = 1$ cette sortie.
- Si la sortie s est égale à l'objectif b à atteindre alors on conserve le neurone tel quel : $[p'_1, p'_2, p'_3] = [p_1, p_2, p_3]$. En effet le neurone a déjà le bon comportement pour cette entrée, on ne change donc rien.
- Si la sortie s est différente de l'objectif, alors on modifie l'état du neurone en $[p'_1, p'_2, p'_3]$ selon la formule suivante :

$$\begin{cases} p'_1 &= p_1 \pm \epsilon e_1 \\ p'_2 &= p_2 \pm \epsilon e_2 \\ p'_3 &= p_3 \pm \epsilon e_3 \end{cases}$$

On prendra $\epsilon = 0.2$ par exemple. Le signe est « + » si l'activation calculée est $s = 0$ alors que l'objectif est $b = 1$. Le signe est « - » si l'activation calculée est $s = 1$ alors que l'objectif est $b = 0$.

- La fonction renvoie $[p'_1, p'_2, p'_3]$.

Exemples. Partons du neurone dont l'état de départ de neurone est $[p_1, p_2, p_3] = [1, 1, 1]$.

- Si l'entrée est $[e_1, e_2, e_3] = [1, 0, 2]$ et l'objectif à atteindre est $b = 1$. On calcule $s = 1$. Comme $s = b$ alors on ne fait rien et $[p'_1, p'_2, p'_3] = [p_1, p_2, p_3]$.
- Ce serait pareil si l'entrée était $[0.5, 0.1, 0.2]$ avec un objectif de $b = 0$ (car on a aussi $s = 0$).
- Si l'entrée est $[e_1, e_2, e_3] = [0.5, 0.2, 0]$ et l'objectif est $b = 1$. On calcule $s = 0$. Comme l'objectif est différent de la sortie, on modifie l'état du neurone, selon la formule $p'_i = p_i + \epsilon e_i$. On trouve donc :

$$p'_1 = 1 + 0.2 \times 0.5 = 1.1 \quad p'_2 = 1 + 0.2 \times 0.2 = 1.04 \quad p'_3 = 1 + 0.2 \times 0 = 1$$

Le nouvel état du neurone est donc $[1.1, 1.04, 1]$.

- On repart de l'état initial du neurone $[p_1, p_2, p_3] = [1, 1, 1]$. Si l'entrée est $[e_1, e_2, e_3] = [0, 1, 1]$ et l'objectif est $b = 0$. On calcule $s = 1$. Comme l'objectif est différent de la sortie on modifie l'état du neurone, selon la formule $p'_i = p_i - \epsilon e_i$. On trouve donc :

$$p'_1 = 1 - 0.2 \times 0 = 1 \quad p'_2 = 1 - 0.2 \times 1 = 0.8 \quad p'_3 = 1 - 0.2 \times 1 = 0.8$$

Indications. Pour éviter les problèmes en modifiant une liste, tu peux commencer par la copier, avec par exemple :

```
nouv_neurone = list(neurone)
```

Tu peux ensuite modifier la liste `nouv_neurone`.

3. Apprentissage itéré.

Pour que le neurone s'entraîne il faut lui procurer plusieurs données. Programme une fonction `epoque_apprentissage(neurone_init, liste_entrees_objectifs)` qui calcule le nouvel état du neurone après entraînement sur tous les éléments de la liste. La liste d'entraînement est une liste composée de couples entrée/objectif attendu :

```
[ ([1,0,0],1), ([0,1,1],0), ([0,1,0],0), ... ]
```

Il s'agit juste d'itérer la fonction `apprentissage()` sur chaque élément de la liste. Un entraînement sur la totalité du jeu de test s'appelle une *époque*.

4. Apprentissage de la couleur rouge.

Entraîne ton neurone afin qu'il reconnaisse la couleur rouge.

Méthode.

- En entrée les paramètres $[e_1, e_2, e_3]$ correspondent au code couleur $[r, g, b]$ (trois nombres réels entre 0 et 1).
- Si la sortie vaut 1 alors le neurone déclare que c'est du rouge, si la sortie vaut 0 il déclare que ce n'est pas du rouge.
- Pars d'un neurone dans un état quelconque, par exemple $[p_1, p_2, p_3] = [1, 1, 1]$ et fait le évoluer par apprentissage sur toute la liste d'entraînement (première époque).
- À partir du nouvel état du neurone, recommence l'apprentissage avec toujours la même liste d'entraînement (deuxième époque, puis troisième époque...).
- Au bout d'une dizaine d'époques l'état du neurone devrait être stable et bien paramétré.

Entraînement. Voici le couple entrée/objectif que tu peux utiliser pour apprendre la couleur rouge.

```
liste_entrees_objectifs = [
    ([1,0,0],1), ([0,1,1],0), ([1,1,0],0),
    ([1,0,0.2],1), ([0,1,0],0), ([0,0,0],0),
    ([1,0,1],0), ([0.7,0,0],1), ([0.5,0.5,0.5],0),
    ([0.9,0.2,0],1), ([0.9,0,0],1), ([1,1,1],0),
    ([0.2,1,0],0), ([0.8,0.2,0],1), ([0.7,0.1,0.1],1) ]
```

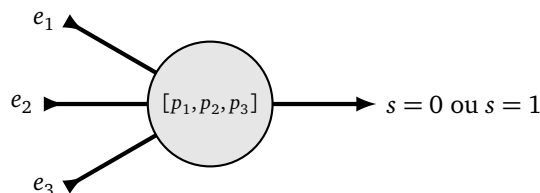
Par exemple le couple $([1, 0, 0], 1)$ signifie que la couleur de code *rgb* $[1, 0, 0]$ est du rouge ; le couple $([0, 1, 1], 0)$ signifie que la couleur de code *rgb* $[0, 1, 1]$ n'est pas du rouge. Note que pour faire comprendre au neurone ce qu'est du rouge, il faut aussi lui montrer des couleurs différentes du rouge. Voici la liste d'apprentissage sous la forme d'un tableau de couleur.

Couleur	Code <i>rgb</i>	Est-ce du rouge ?
	(1, 0, 0)	oui
	(1, 0, 0.2)	oui
	(0.7, 0, 0)	oui
	(0.9, 0.2, 0)	oui
	(0.8, 0.2, 0)	oui
	(0.9, 0, 0)	oui
	(0.7, 0.1, 0.1)	oui

Couleur	Code <i>rgb</i>	Est-ce du rouge ?
	(0, 1, 1)	non
	(1, 1, 0)	non
	(0, 1, 0)	non
	(0, 0, 0)	non
	(1, 0, 1)	non
	(0.5, 0.5, 0.5)	non
	(1, 1, 1)	non
	(0.2, 1, 0)	non

Liste d'entraînement de la couleur rouge.

Résultats. En partant de l'état initial $[p_1, p_2, p_3] = [1, 1, 1]$ avec un pas de $\epsilon = 0.2$ alors l'état du neurone converge en 10 époques vers $[1.66, -0.78, -0.66]$ (ces valeurs dépendent de l'échantillon d'apprentissage, de ϵ et du nombre d'époques).



Vérifications. Vérifions que notre neurone, avec l'état $[p_1, p_2, p_3] = [1.66, -0.78, -0.66]$, détecte correctement le rouge, c'est-à-dire que la fonction activation() renvoie 1 uniquement pour une couleur rouge en entrée. Voici les résultats obtenus :

Couleur	Code <i>rgb</i>	$q = p_1 e_1 + p_2 e_2 + p_3 e_3$	Sortie <i>s</i>	Rouge ?
	(0.9, 0, 0)	1.49	1	oui
	(1, 0.2, 0.2)	1.37	1	oui
	(0, 0, 1)	-0.66	0	non
	(1, 0.5, 0)	1.27	1	oui
	(0.7, 0.5, 0.4)	0.50	0	non

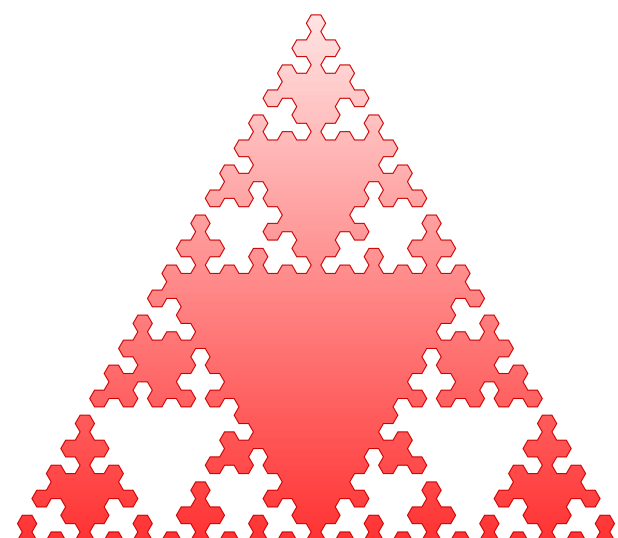
Détection du rouge par le neurone.

Notes.

- Notre neurone fonctionne remarquablement bien !
- Cependant il considère le orange comme du rouge (les goûts et les couleurs...).
- Remarque aussi que l'on n'a jamais montré au neurone la couleur bleue lors de l'apprentissage, mais au final il sait que le bleu n'est pas du rouge !

Projet. Programme une interface graphique qui permet l'apprentissage interactif en proposant à l'utilisateur des couleurs avec un choix « rouge/pas rouge » ; affiche aussi l'état du neurone, son évolution et sa réussite sur un jeu de test.

QUATRIÈME PARTIE

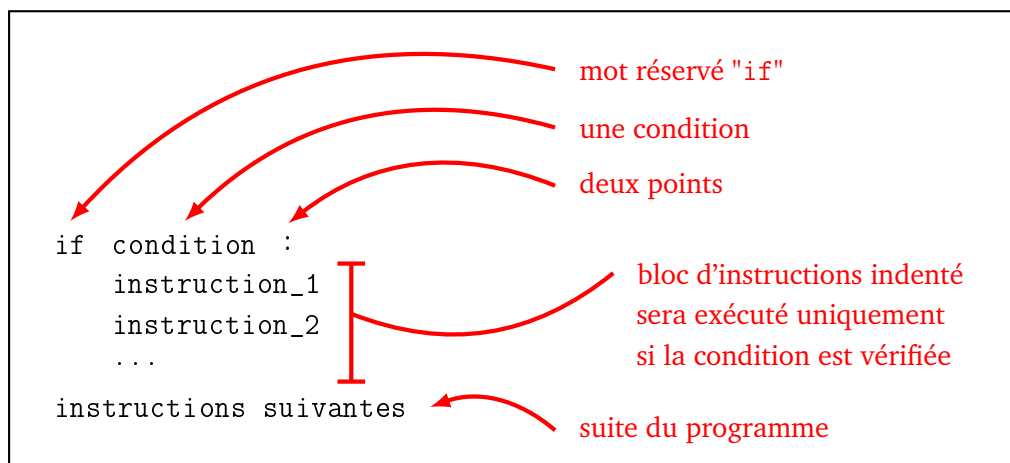


GUIDES

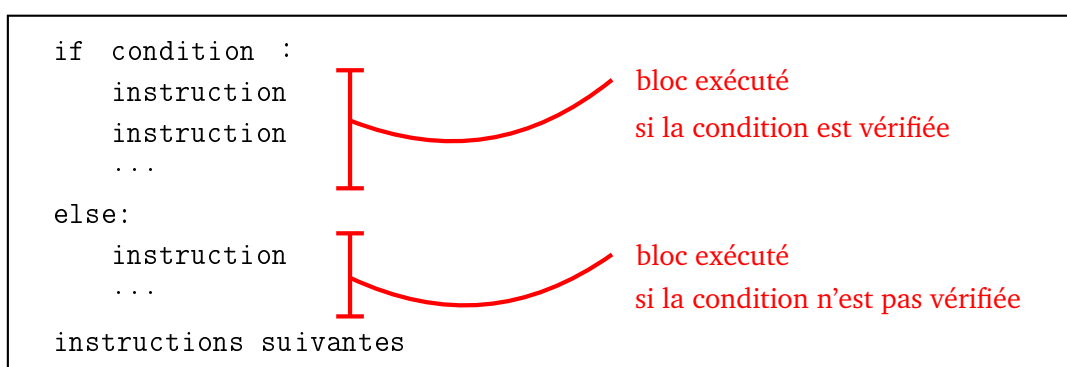
Guide de survie Python

1. Test et boucles

1.1. Si ... alors ...



1.2. Si ... alors ... sinon ...



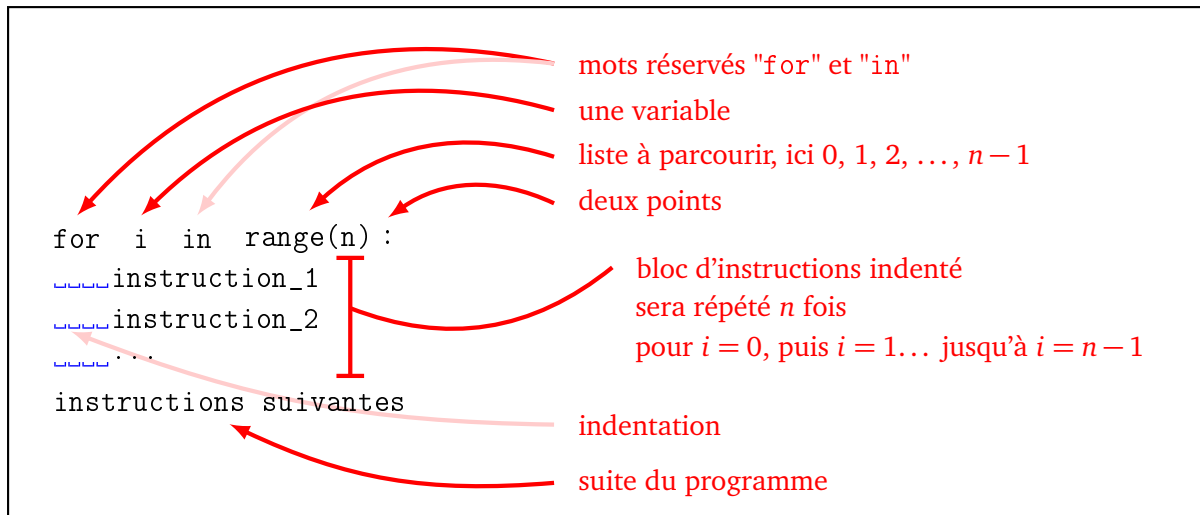
1.3. Sinon si ...

Il est possible d'enchaîner plusieurs tests avec des instructions `elif` qui correspondent à des « sinon si ». Voici un exemple avec un entier n à tester :

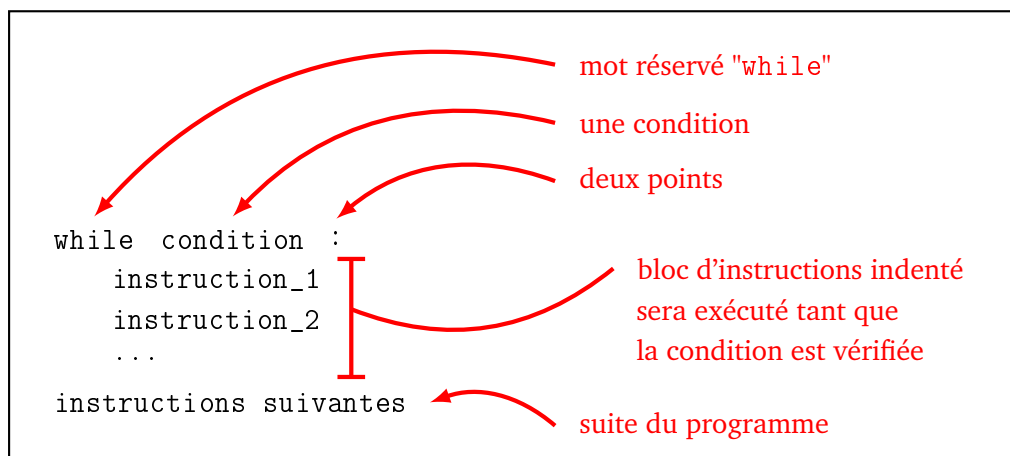
```
if n < 0:  
    print("Le nombre est négatif.")  
elif n == 0:
```

```
print("Le nombre est nul.")
elif 1 <= n < 10:
    print("Le nombre est un chiffre non nul.")
else:
    print("Le nombre est plus grand que 10.")
```

1.4. Boucle pour



1.5. Boucle tant que



1.6. Quitter une itération ou une boucle

- La commande Python pour quitter immédiatement une boucle « tant que » ou une boucle « pour » est l'instruction `break`. Le programme passe immédiatement aux instructions d'après la boucle.
- La commande `continue` interrompt l'itération en cours (sans quitter la boucle) et passe immédiatement à l'itération suivante.

1.7. Ne rien faire !

La commande `pass` ne fait rien. C'est utile pour avoir du code à compléter plus tard mais dont la syntaxe est déjà correcte.

```
if n == 0:
    pass    # je traiterai ce cas particulier quand j'aurai le temps !
else:
    moyenne = somme/n
```

2. Type de données

Principaux types

- `int` Entier. Exemples : 123 ou -15.
- `float` Nombre flottant (ou à virgule). Exemples : 4.56, -0.001, 6.022e23 (pour 6.022×10^{23}), 4e-3 (pour $0.004 = 4 \times 10^{-3}$).
- `complex` Nombre complexe flottant. Le caractère `j` correspond au nombre complexe i . Exemples : $1+2j$ (pour $1+2i$), $1/(3-1j)$ pour $\frac{1}{3-i}$.
- `str` Caractère ou chaîne de caractères. Exemples : 'Y', "k", 'Hello', "World !".
- `bool` Booléen. True ou False.
- `list` Liste. Exemple : [1,2,3,4].
- `tuple` Liste immuable (ne peut être changée). Exemple : (1,2,3,4).
- `dict` Dictionnaire. Exemple :
`grandes_dates = {'marignan':1515, 'revolution':1789 , 'waterloo':1815}`

Connaître le type

La fonction `type()` renvoie le type d'un élément. Par exemple `type(5)` renvoie `<class 'int'>`, mais l'utilisation courante se fait de la façon suivante :

- `type(5) == int` renvoie « Vrai »,
- `type(5.5) == int` renvoie « Faux ».

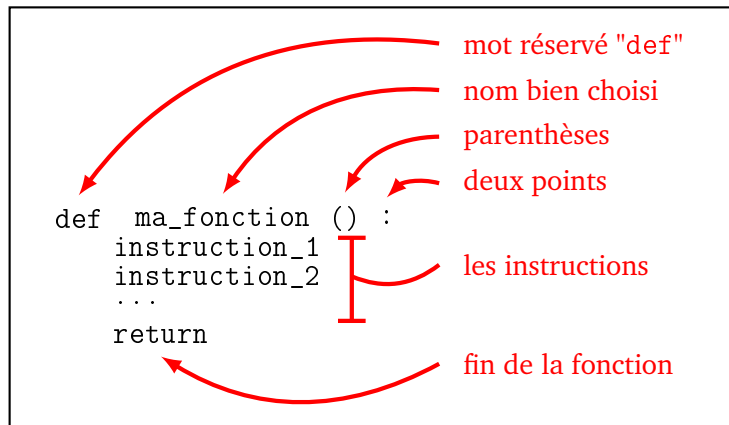
Vérifier le type

Pour savoir si un élément est d'un type donné tu peux utiliser la fonction `isinstance(element, type)`. Par exemple :

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

3. Définir des fonctions

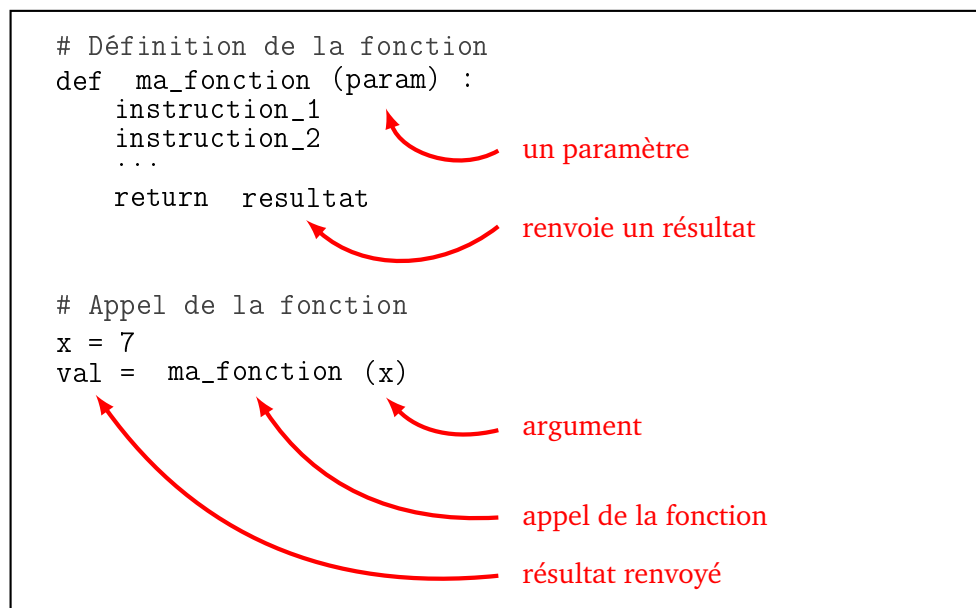
3.1. Définition d'une fonction



3.2. Fonction avec paramètre

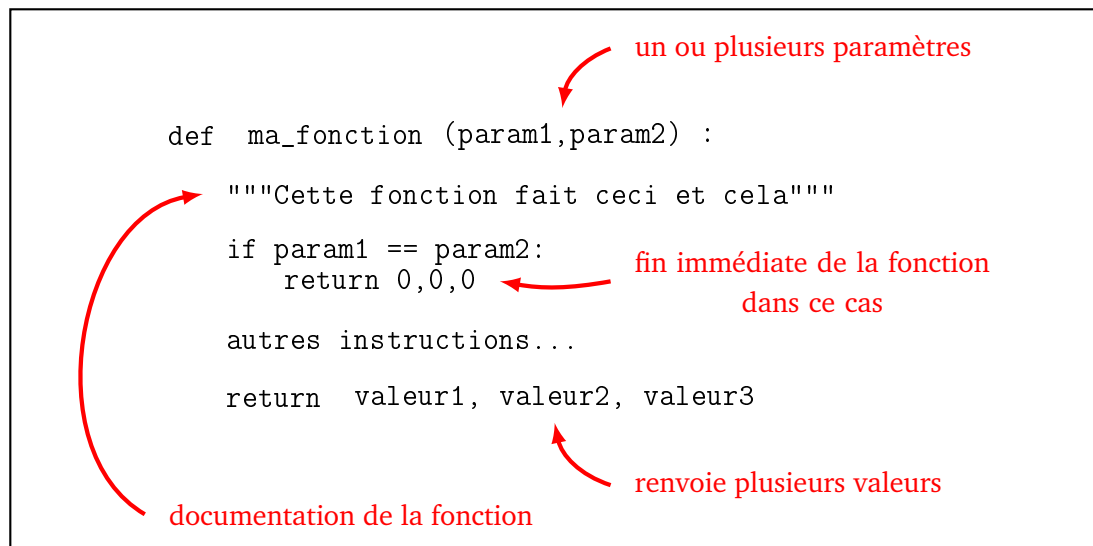
Les fonctions informatiques acquièrent tout leur potentiel avec :

- une *entrée*, qui regroupe des variables qui servent de *paramètres*,
- une *sortie*, qui est un résultat renvoyé par la fonction (et qui souvent dépendra des paramètres d'entrée).



3.3. Fonction avec plusieurs paramètres

Il peut y avoir plusieurs paramètres en entrée, il peut y avoir plusieurs résultats en sortie.



Voici un exemple d'une fonction avec deux paramètres et deux sorties.

```
def somme_produit(x,y):
    """ Calcule la somme et le produit de deux nombres. """
    S = x + y      # Somme
    P = x*y        # Produit
    return S, P    # Renvoie les résultats

# Appel de la fonction
som, prod = somme_produit(3,7) # Résultats
print("Somme :",som)          # Affichage
print("Produit :",prod)        # Affichage
```

- Très important! Il ne faut pas confondre afficher et renvoyer une valeur. L'affichage (par la commande `print()`) affiche juste quelque chose à l'écran. La plupart des fonctions n'affichent rien, mais renvoient une valeur (ou plusieurs). C'est beaucoup plus utile car cette valeur peut être utilisée ailleurs dans le programme.
- Dès que le programme rencontre l'instruction `return`, la fonction s'arrête et renvoie le résultat. Il peut y avoir plusieurs fois l'instruction `return` dans une fonction mais une seule sera exécutée. On peut aussi ne pas mettre d'instruction `return` si la fonction ne renvoie rien.
- Dans les instructions d'une fonction, on peut bien sûr faire appel à d'autres fonctions!

3.4. Commentaires et docstring

- **Commentaire.** Tout ce qui suit le signe dièse `#` est un commentaire et est ignoré par Python. Par exemple :

```
# Boucle principale
while r != 0:    # Tant que le reste n'est pas nul
    r = r - 1    # Diminuer le reste
```

- **Docstring.** Tu peux décrire ce que fait une fonction en commençant par un *docstring*, c'est-à-dire une description en français, entourée par trois guillemets. Par exemple :

```
def produit(x,y):
    """ Calcule le produit de deux nombres
    Entrée : deux nombres x et y
    Sortie : le produit de x par y """
```

```
p = x * y
return p
```

3.5. Variable locale

Voici une fonction toute simple qui prend en entrée un nombre et renvoie le nombre augmenté de un.

```
def ma_fonction(x):
    x = x + 1
    return x
```

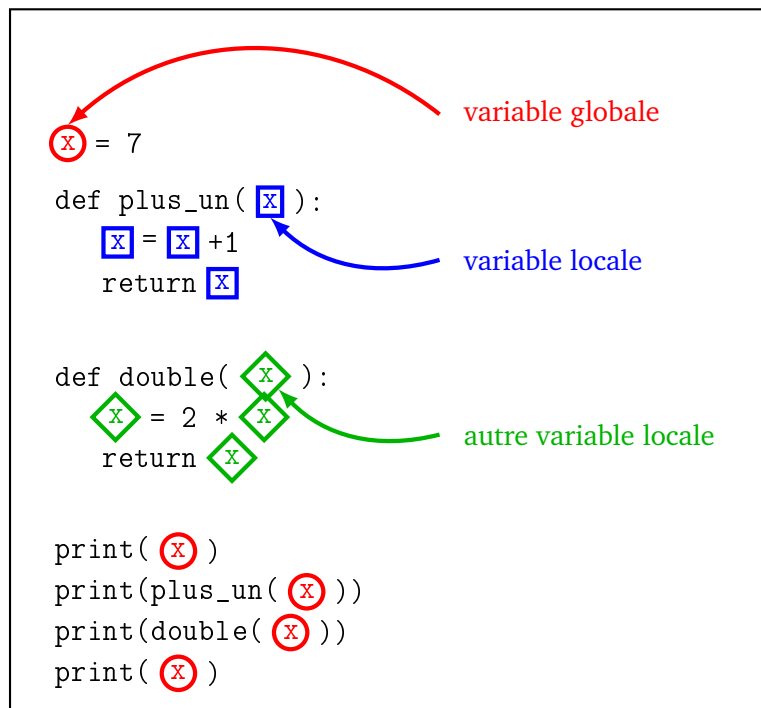
- Bien évidemment `ma_fonction(3)` renvoie 4.
- Si la valeur de `y` est 5, alors `ma_fonction(y)` renvoie 6. Mais attention, la valeur de `y` n'a pas changé, elle vaut toujours 5.
- Voici la situation problématique qu'il faut bien comprendre :

```
x = 7
print(ma_fonction(x))
print(x)
```

- La variable `x` est initialisée à 7.
- L'appel de la fonction `ma_fonction(x)` est donc la même chose que `ma_fonction(7)` et renvoie logiquement 8.
- Que vaut la variable `x` à la fin ? La variable `x` est inchangée et vaut toujours 7 ! Même s'il y a eu entre temps une instruction `x = x + 1`. Cette instruction a changé le `x` à l'intérieur de la fonction, mais pas le `x` en dehors de la fonction.

- Les variables définies à l'intérieur d'une fonction sont appelées **variables locales**. Elles n'existent pas en dehors de la fonction.
- Si une variable dans une fonction porte le même nom qu'une variable dans le programme (comme le `x` dans l'exemple ci-dessus), il y a deux variables distinctes ; la variable locale n'existant que dans la fonction.

Pour bien comprendre la portée des variables, tu peux colorier les variables globales d'une fonction en rouge, et les variables locales avec une couleur par fonction. Le petit programme suivant définit une fonction qui ajoute un, et une autre qui calcule le double.



Le programme affiche d'abord la valeur de `x`, donc 7, puis il ajoute un à 7, il affiche donc 8, puis il affiche le double de `x`, donc 14. La variable globale `x` n'a jamais changé, le dernier affichage de `x` est donc encore 7.

3.6. Variable globale

Une **variable globale** est une variable qui est définie pour l'ensemble du programme. Il n'est généralement pas recommandé d'utiliser de telles variables, mais cela peut être utile dans certains cas. Voyons un exemple. On déclare la variable globale, ici la constante de gravitation, en début de programme comme une variable classique :

```
gravitation = 9.81
```

La contenu de la variable `gravitation` est maintenant accessible partout. Par contre, si on souhaite changer la valeur de cette variable dans une fonction, il faut bien préciser à Python que l'on est conscient de modifier une variable globale !

Par exemple pour des calculs sur la Lune, il faut changer la constante de gravitation qui y est beaucoup plus faible.

```
def sur_la_lune():
    global gravitation    # Oui, je veux modifier cette variable globale!
    gravitation = 1.625   # Nouvelle valeur pour tout le programme
    ...
```

3.7. Arguments optionnels

Il est possible de donner des arguments optionnels. Voici comment définir une fonction (ici qui dessinerait un trait) en donnant des valeurs par défaut :

```
def tracer(longueur, epaisseur=5, couleur="blue"):
```

- La commande `tracer(100)` trace mon trait, et comme je n'ai précisé que la longueur, les arguments `epaisseur` et `couleur` prennent les valeurs par défaut (5 et bleu).
- La commande `tracer(100, epaisseur=10)` trace mon trait avec une nouvelle épaisseur (la couleur est celle par défaut).

- La commande `tracer(100, couleur="red")` trace mon trait avec une nouvelle couleur (l'épaisseur est celle par défaut).
- La commande `tracer(100, epaisseur=10, couleur="red")` trace mon trait avec une nouvelle épaisseur et une nouvelle couleur.
- Voici aussi ce que tu peux utiliser :
 - `tracer(100, 10, "red")` : ne pas préciser les noms des options si on fait attention à l'ordre.
 - `tracer(couleur="red", epaisseur=10, longueur=100)` : on peut nommer n'importe quelle variable ; les variables nommées peuvent être passées en paramètre dans n'importe quel ordre !

3.8. Fonction lambda

Une fonction lambda (lettre grecque λ) est une façon simple de définir une fonction en Python. Par exemple :

```
f = lambda x: x**2
```

correspond à la fonction $f : x \mapsto x^2$ et est une alternative condensée au code suivant :

```
def f(x):
    return x**2
```

Une fonction est un objet Python comme un autre. Elle peut donc être utilisée dans le programme comme dans l'exemple suivant qui teste si $f(a) > f(b)$:

```
def est_plus_grand(f, a, b):
    if f(a) > f(b):
        return True
    else:
        return False
```

Pour les deux fonctions f définies au-dessus (soit à l'aide de `lambda`, soit à l'aide de `def`) alors

```
est_plus_grand(f, 1, 2)
```

renvoie « Faux ».

À l'aide des fonctions lambda on peut aussi se permettre de ne pas donner de nom à une fonction, comme ci-dessous avec la fonction $x \mapsto \frac{1}{x}$. Alors

```
est_plus_grand(lambda x: 1/x, 1, 2)
```

qui renvoie « Vrai » (`lambda x: 1/x` joue le rôle de f).

4. Modules

4.1. Utiliser un module

- `from math import *` Importe toutes les fonctions du module `math`. Pour pouvoir utiliser par exemple la fonction sinus par `sin(0)`. C'est la méthode la plus simple et c'est celle que nous utilisons dans ce livre.
- `import math` Permet d'utiliser les fonctions du module `math`. On a alors accès à la fonction sinus par `math.sin(0)`. C'est la méthode recommandée officiellement afin d'éviter les conflits entre les modules.

4.2. Principaux modules

- `math` contient les principales fonctions mathématiques.
- `cmath` contient les fonctions mathématiques pour les nombres complexes.
- `random` simule le tirage au hasard.
- `turtle` la tortue Python, l'équivalent de *Scratch*.
- `matplotlib` permet de tracer des graphiques et visualiser des données.
- `tkinter` permet d'afficher des fenêtres graphiques.
- `time` pour l'heure, la date et chronométrer.
- `timeit` pour mesurer le temps d'exécution d'une fonction.

Il existe beaucoup d'autres modules !

5. Erreurs

5.1. Erreurs d'indentation

```
a = 3
b = 2
```

Python renvoie le message d'erreur *IndentationError : unexpected indent*. Il indique le numéro de ligne où se situe l'erreur d'indentation, il pointe même à l'aide du symbole « ^ » l'endroit exact de l'erreur.

5.2. Erreurs de syntaxe

- ```
while x >= 0
 x = x - 1
```

Python renvoie le message d'erreur *SyntaxError : invalid syntax* car il manque les deux points après la condition `while x >= 0 :`

- `chaine = Coucou le monde` renvoie une erreur car il manque les guillemets pour définir la chaîne de caractères.
- `print("Coucou"` Python renvoie le message d'erreur *SyntaxError : unexpected EOF while parsing* car l'expression est mal parenthésée.
- `if val = 1:` Encore une erreur de syntaxe, car il faudrait écrire `if val == 1:`.

### 5.3. Erreurs de type

- **Entier**

```
n = 7.0
for i in range(n):
 print(i)
```

Python renvoie le message d'erreur *TypeError : 'float' object cannot be interpreted as an integer*. En effet 7.0 n'est pas un entier, mais un nombre flottant.

- **Nombre flottant**

```
x = "9"
sqrt(x)
```

Python renvoie le message d'erreur *TypeError : a float is required*, car "9" est une chaîne de caractères et pas un nombre.

- **Mauvais nombre d'arguments**

`gcd(12)` Python renvoie le message d'erreur *TypeError : gcd() takes exactly 2 arguments (1 given)* car la fonction `gcd()` du module `math` a besoin des deux arguments, comme par exemple `gcd(12,18)`.

## 5.4. Erreurs de nom

- `if y != 0: y = y - 1` Python renvoie le message *NameError : name 'y' is not defined* si la variable `y` n'a pas encore été définie.
- Cette erreur peut aussi se produire si les minuscules/majuscules ne pas scrupuleusement respectées. `variable`, `Variable` et `VARIABLE` sont trois noms de variables différents.
- `x = sqrt(2)` Python renvoie le message *NameError : name 'sqrt' is not defined*, il faut importer le module `math` pour pouvoir utiliser la fonction `sqrt()`.
- **Fonction non encore définie**

```
produit(6,7)
```

```
def produit(a,b):
 return a*b
```

Renvoie une erreur *NameError : name 'produit' is not defined* car une fonction doit être définie avant d'être utilisée.



## 6. Programmation objet

The diagram illustrates the structure and usage of a Python class named `Vecteur`. It includes the class definition with methods `__init__`, `__str__`, `norme`, `produit_par_scalaire`, `addition`, and `__add__`. Below the class definition, two examples of object creation and usage are shown. Red arrows point from descriptive text to specific parts of the code.

```
class Vecteur:
 def __init__(self, x, y, z):
 self.x = x
 self.y = y
 self.z = z

 def __str__(self):
 ligne = "(" + str(self.x) + "," + str(self.y) + "," + str(self.z) + ")"
 return ligne

 def norme(self):
 N = sqrt(self.x**2 + self.y**2 + self.z**2)
 return N

 def produit_par_scalaire(self, k):
 W = Vecteur(k*self.x, k*self.y, k*self.z)
 return W

 def addition(self, other):
 W = Vecteur(self.x+other.x, self.y+other.y, self.z+other.z)
 return W

 def __add__(self, other):
 W = Vecteur(self.x+other.x, self.y+other.y, self.z+other.z)
 return W
```

**Annotations for the class definition:**

- `class`: mot réservé class
- `Vecteur`: nom de la classe
- `def __init__(self, x, y, z):`: méthode d'initialisation `__init__()`
- `self`: self correspond à l'objet en cours
- `x, y, z`: définition des attributs x, y, z
- `def __str__(self):`: méthode pour l'affichage par `print()`
- `self`: self : objet en cours
- `self.x`: self.x : valeur de l'attribut x de l'objet en cours
- `return N`: renvoie un nombre
- `def produit_par_scalaire(self, k):`: définition d'un objet de la classe `Vecteur`
- `return W`: renvoie un objet
- `def addition(self, other):`: self : objet en cours; other : un autre objet
- `return W`: renvoie un nouvel objet
- `def __add__(self, other):`: méthode `__add__()` pour l'addition par "+"

**Annotations for Example 1:**

- `V = Vecteur(1,2,3)`: un objet V : une instance de la classe `Vecteur` initialisée par des valeurs x, y, z
- `print("Valeur de x :", V.x)`: V.x valeur de l'attribut x associé à V
- `print("Vecteur :", V)`: affichage de l'objet V grâce à la méthode `__str__()`
- `print("Norme :", V.norme())`: appel de la méthode `norme()`; l'argument V correspond au paramètre `self`

**Annotations for Example 2:**

- `V1 = Vecteur(1,2,3)` and `V2 = Vecteur(1,0,-4)`: définition de deux objets
- `V3 = V1.addition(V2)`: appel de la méthode `addition()`; l'argument V1 correspond au paramètre `self`; l'argument V2 correspond au paramètre `other`
- `V4 = V1 + V2`: utilisation de "+" par l'appel à la méthode `__add__()`

# Principales fonctions

## 1. Mathématiques

### Opérations classiques

- $a + b$ ,  $a - b$ ,  $a * b$  opérations classiques
- $a / b$  division « réelle » (renvoie un nombre flottant)
- $a // b$  quotient de la division euclidienne (renvoie un entier)
- $a \% b$  reste de la division euclidienne, appelé  $a$  modulo  $b$
- $\text{abs}(x)$  valeur absolue
- $x ** n$  puissance  $x^n$
- $4.56\text{e}12$  pour  $4.56 \times 10^{12}$

### Module « math »

L'usage d'autres fonctions mathématiques nécessite le recours au module `math` qui s'appelle par la commande :

```
from math import *
```

- $\text{sqrt}(x)$  racine carrée  $\sqrt{x}$
- $\cos(x)$ ,  $\sin(x)$ ,  $\tan(x)$  fonctions trigonométriques  $\cos x$ ,  $\sin x$ ,  $\tan x$  en radians
- `pi` valeur approchée de  $\pi = 3.14159265\dots$
- `inf` valeur  $+\infty$  (plus grande que tout autre nombre flottant)
- `-inf` valeur  $-\infty$  (plus petite que tout autre nombre flottant)
- $\text{floor}(x)$  entier juste en-dessous de  $x$
- $\text{ceil}(x)$  entier juste au-dessus de  $x$
- $\text{gcd}(a, b)$  pgcd de  $a$  et de  $b$
- $\text{exp}(x)$  exponentielle  $e^x$
- $\log(x)$  logarithme népérien (en base  $e$ ),  $\ln(x)$
- $\log(x, b)$  logarithme de  $x$  en base  $b$ ,  $\log_b(x)$
- $\log(x, 10)$  logarithme décimal,  $\log_{10}(x)$
- $\log(x, 2)$  logarithme en base 2,  $\log_2(x)$
- $\text{acos}(x)$ ,  $\text{asin}(x)$ ,  $\text{atan}(x)$  fonctions trigonométriques inverse  $\arccos x$ ,  $\arcsin x$ ,  $\arctan x$ , renvoie un angle en radians
- $\text{atan2}(y, x)$  renvoie l'angle  $\arctan(\frac{y}{x})$  en radians. C'est l'angle entre l'horizontale et le vecteur  $\overrightarrow{OM}$ , où  $M$  est le point de coordonnées  $(x, y)$ .

### Nombres complexes

Les nombres complexes sont compris nativement par Python, cependant pour davantage de fonctionnalités on peut utiliser le module `cmath`. Pour éviter les conflits avec le module `math` on l'importe par :

```
import cmath
```

- `z.real` (sans parenthèses) partie réelle  $a$  de  $z = a + ib$
- `z.imag` (sans parenthèses) partie imaginaire  $b$  de  $z = a + ib$
- `abs(z)` module  $|z| = \sqrt{a^2 + b^2}$
- `z.conjugate()` conjugué  $\bar{z} = a - ib$
- `cmath.phase(z)` argument  $\theta \in ]-\pi, +\pi]$  de  $z$
- `cmath.rect(r, theta)` renvoie le nombre complexe dont le module est  $r$  et l'argument  $\theta$

### Module « random »

Le module `random` génère des nombres de façon pseudo-aléatoire. Il s'appelle par la commande :

```
from random import *
```

- `random()` à chaque appel, renvoie un nombre flottant  $x$  au hasard vérifiant  $0 \leq x < 1$ .
- `randint(a,b)` à chaque appel, renvoie un nombre entier  $n$  au hasard vérifiant  $a \leq n \leq b$ .
- `choice(liste)` à chaque appel, tire au hasard un élément de la liste.
- `liste.shuffle()` mélange la liste (la liste est modifiée).

### Écriture binaire

- `bin(n)` renvoie l'écriture binaire de l'entier  $n$  sous la forme d'une chaîne. Exemple : `bin(17)` renvoie `'0b10001'`.
- Pour écrire directement un nombre en écriture binaire, il suffit d'écrire le nombre en commençant par `0b` (sans guillemets). Par exemple `0b11011` vaut 27.

### Affectation multiple

Python permet les affectations multiples, ce qui permet d'échanger facilement le contenu de deux variables.

- **Affectation multiple.**

```
a, b, c = 3, 4, 5
```

Maintenant `a` vaut 3, `b` vaut 4 et `c` vaut 5.

- **Échange de valeurs.**

```
a, b = b, a
```

Maintenant `a` vaut l'ancien contenu de `b` donc vaut 4 et `b` vaut l'ancien contenu de `a` donc 3.

- **Échange à la main.** Pour échanger deux valeurs sans utiliser la double affectation, il faut introduire une variable temporaire :

```
temp = a
a = b
b = temp
```

*Note.* Python est suffisamment intelligent pour autoriser une syntaxe souple, par exemple afin de passer d'une liste aux éléments de cette liste :

```
liste = [2,3,4]
a,b,c = liste # a vaut 2, b vaut 3,...
```

Par contre lors d'un appel à une fonction il est nécessaire de décompacter (*unpacking*) à l'aide de l'opérateur «`*`» utilisé en préfixe.

```
def ma_fonction(a,b,c):
 ...
```

```
liste = [2,3,4]
ma_fonction(*liste) # note l'étoile !
```

L'instruction `ma_fonction(*liste)` équivaut ici à l'appel `ma_fonction(2,3,4)`.

### Incrémentation rapide

Pour incrémenter une variable on écrit simplement :

$$x = x + 1$$

mais on peut utiliser l'opérateur « += » pour écrire plus simplement :

$$x += 1$$

.

L'opérateur « += » peut être utilisé dans d'autres situations :

- `x += 2` pour `x = x + 2`
- `chaine += "mot"` pour `chaine = chaine + "mot"`
- `liste += [element]` pour `liste = liste + [element]` ou `liste.append(element)`

### Évaluation

- `eval(chaine)` permet d'évaluer une expression donnée sous la forme d'une chaîne de caractères.
- Par exemple `eval('8*3')` renvoie 24.
- Par exemple `eval('2+2 == 2*2')` renvoie True.

## 2. Booléens

Un booléen est une donnée qui prend soit la valeur True (« Vrai »), soit la valeur False (« Faux »).

### Comparaisons

Les tests de comparaison suivants renvoient un booléen.

- `a == b` test d'égalité
- `a < b` test inférieur strict
- `a <= b` test inférieur large
- `a > b` ou `a >= b` test supérieur
- `a != b` test de non égalité

Ne pas confondre « `a = b` » (affectation) et « `a == b` » (test d'égalité).

### Opérations sur les booléens

- `P and Q` « et » logique
- `P or Q` « ou » logique
- `not P` négation

## 3. Chaînes de caractères I

### Chaînes

- `"A"` ou `'A'` un caractère
- `"Python"` ou `'Python'` une chaîne de caractères
- `len(chaine)` la longueur de la chaîne. Exemple : `len("Python")` renvoie 6.
- `chaine1 + chaine2` concaténation.  
Exemple : `"J aime bien" + "Python"` renvoie `"J aime bienPython"`.
- `chaine[i]` renvoie le *i*-ème caractère de `chaine` (la numérotation commence à 0).  
Exemple avec `chaine = "Python"`, `chaine[1]` vaut `"y"`. Voir le tableau ci-dessous.

| Lettre | P | y | t | h | o | n |
|--------|---|---|---|---|---|---|
| Rang   | 0 | 1 | 2 | 3 | 4 | 5 |

### Conversion nombre/chaîne

- **Chaîne.** `str(nombre)` convertit un nombre (entier ou flottant) en une chaîne. Exemples : `str(7)` renvoie la chaîne "7"; `str(1.234)` renvoie la chaîne "1.234".
- **Entier.** `int(chaine)` renvoie l'entier correspondant à la chaîne. Exemple `int("45")` renvoie l'entier 45.
- **Nombre flottant.** `float(chaine)` renvoie le nombre flottant correspondant à la chaîne. Exemple `float("3.14")` renvoie le nombre 3.14.

### Sous-chaînes

- `chaine[i:j]` renvoie la sous-chaîne des caractères de rang  $i$  à  $j-1$  de `chaine`.  
Exemple : avec `chaine = "Ceci est une chaine"`, `chaine[2:6]` renvoie "ci e".
- `chaine[i:]` renvoie les caractères de rang  $i$  jusqu'à la fin de `chaine`.  
Exemple : `chaine[5:]` renvoie "est une chaine".
- `chaine[:j]` renvoie les caractères du début jusqu'au rang  $j-1$  de `chaine`. Exemple : `chaine[:4]` renvoie "Ceci".

### Mise en forme

La méthode `format()` permet de mettre en forme du texte ou des nombres. Cette fonction renvoie une chaîne de caractères.

- **Texte**

```

TestTestTest
— '{:10}'.format('Test') alignement à gauche (sur 10 caractères)
— '{:>10}'.format('Test') alignement à droite
— '{:^10}'.format('Test') centré

```

- **Entier**

```

456456000456
— '{:d}'.format(456) entier
— '{:6d}'.format(456) alignement à droite (sur 6 caractères)
— '{:06d}'.format(456) ajout de zéros non significatifs (sur 6 caractères)

```

- **Nombre flottant**

```

3.1415933.141592653.1416003.1416
— '{:f}'.format(3.141592653589793) nombre flottant
— '{:.8f}'.format(3.141592653589793) 8 chiffres après la virgule
— '{:8.4f}'.format(3.141592653589793) sur 8 caractères avec 4 chiffres après la virgule
— '{:08.4f}'.format(3.141592653589793) ajout de zéros non significatifs

```

## 4. Chaînes de caractères II

### Encodage

- `chr(n)` renvoie le caractère associé au numéro de code ASCII/unicode  $n$ . Exemple : `chr(65)` renvoie "A"; `chr(97)` renvoie "a".
- `ord(c)` renvoie le numéro de code ASCII/unicode associé au caractère  $c$ . Exemple : `ord("A")` renvoie 65; `ord("a")` renvoie 97.

Le début de la table des codes ASCII/unicode est donné ci-dessous.

|    |    |    |   |    |   |    |   |    |   |    |   |     |   |     |   |     |   |     |   |
|----|----|----|---|----|---|----|---|----|---|----|---|-----|---|-----|---|-----|---|-----|---|
| 33 | !  | 43 | + | 53 | 5 | 63 | ? | 73 | I | 83 | S | 93  | ] | 103 | g | 113 | q | 123 | { |
| 34 | "  | 44 | , | 54 | 6 | 64 | @ | 74 | J | 84 | T | 94  | ^ | 104 | h | 114 | r | 124 |   |
| 35 | #  | 45 | - | 55 | 7 | 65 | A | 75 | K | 85 | U | 95  | _ | 105 | i | 115 | s | 125 | } |
| 36 | \$ | 46 | . | 56 | 8 | 66 | B | 76 | L | 86 | V | 96  | ' | 106 | j | 116 | t | 126 | ~ |
| 37 | %  | 47 | / | 57 | 9 | 67 | C | 77 | M | 87 | W | 97  | a | 107 | k | 117 | u | 127 | - |
| 38 | &  | 48 | 0 | 58 | : | 68 | D | 78 | N | 88 | X | 98  | b | 108 | l | 118 | v |     |   |
| 39 | '  | 49 | 1 | 59 | ; | 69 | E | 79 | O | 89 | Y | 99  | c | 109 | m | 119 | w |     |   |
| 40 | (  | 50 | 2 | 60 | < | 70 | F | 80 | P | 90 | Z | 100 | d | 110 | n | 120 | x |     |   |
| 41 | )  | 51 | 3 | 61 | = | 71 | G | 81 | Q | 91 | [ | 101 | e | 111 | o | 121 | y |     |   |
| 42 | *  | 52 | 4 | 62 | > | 72 | H | 82 | R | 92 | \ | 102 | f | 112 | p | 122 | z |     |   |

### Majuscules/minuscules

- `chaine.upper()` renvoie une chaîne en majuscules.
- `chaine.lower()` renvoie une chaîne en minuscules.

### Chercher/remplacer

- `sous_chaine in chaine` renvoie « vrai » ou « faux » si `sous_chaine` apparaît dans `chaine`.  
Exemple : `"PAS" in "ETRE OU NE PAS ETRE"` vaut `True`.
- `chaine.find(sous_chaine)` renvoie le rang auquel la sous-chaîne a été trouvée (et -1 sinon).  
Exemple : avec `chaine = "ABCDE"`, `chaine.find("CD")` renvoie 2.
- `chaine.replace(sous_chaine,nouv_sous_chaine)` remplace chaque occurrence de la sous-chaîne par la nouvelle sous-chaîne.  
Exemple : avec `chaine = "ABCDE"`, `chaine.replace("CD","XY")` renvoie `"ABXYE"`.
- `ligne.strip()` renvoie la chaîne de caractères de la ligne sans les espaces de début et de fin, ni le saut de ligne.  
Exemple : avec `ligne = " Il était une fois ! \n"`, `ligne.strip()` renvoie `'Il était une fois !'`.

### Séparer/regrouper

- `chaine.split(separateur)` sépare la chaîne en une liste de sous-chaînes (par défaut le séparateur est l'espace).  
Exemples :  
— `"Etre ou ne pas etre.".split()` renvoie `['Etre', 'ou', 'ne', 'pas', 'etre.']`  
— `"12.5;17.5;18".split(";")` renvoie `['12.5', '17.5', '18']`
- `separateur.join(liste)` regroupe les sous-chaînes en une seule chaîne en ajoutant le séparateur entre chaque.  
Exemples :  
— `"".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etreounepasetre.'` Il manque les espaces.  
— `" ".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etre ou ne pas etre.'`  
C'est mieux lorsque le séparateur est une espace.

```
— "--".join(["Etre", "ou", "ne", "pas", "etre."]) renvoie
'Etre--ou--ne--pas--etre.'
```

## 5. Listes I

### Construction d'une liste

Exemples :

- `liste1 = [5,4,3,2,1]` une liste de 5 entiers.
- `liste2 = ["Vendredi", "Samedi", "Dimanche"]` une liste de 3 chaînes.
- `liste3 = []` la liste vide.
- `list(range(n))` liste des entiers de 0 à  $n-1$ .
- `list(range(a,b))` liste des entiers de  $a$  à  $b-1$ .
- `list(range(a,b,saut))` liste des entiers de  $a$  à  $b-1$ , avec un pas donné par l'entier saut.

### Accéder à un élément

- `liste[i]` renvoie l'élément de la liste de rang  $i$ . Attention, le rang commence à 0.  
Exemple : `liste = ["A", "B", "C", "D", "E", "F"]` alors `liste[2]` renvoie "C".

| Lettre | "A" | "B" | "C" | "D" | "E" | "F" |
|--------|-----|-----|-----|-----|-----|-----|
| Rang   | 0   | 1   | 2   | 3   | 4   | 5   |

- `liste[-1]` renvoie le dernier élément, `liste[-2]` renvoie l'avant-dernier élément...
- `liste.pop()` supprime le dernier élément de la liste et le renvoie (c'est l'opération « dépiler »).

### Ajouter un élément (ou plusieurs)

- `liste.append(element)` ajoute l'élément à la fin de la liste. Exemple : si `liste = [5,6,7,8]` alors `liste.append(9)` rajoute 9 à la liste, `liste` vaut `[5,6,7,8,9]`.
- `nouv_liste = liste + [element]` fournit une nouvelle liste avec un élément en plus à la fin. Exemple : `[1,2,3,4] + [5]` vaut `[1,2,3,4,5]`.
- `[element] + liste` renvoie une liste où l'élément est ajouté au début. Exemple : `[5] + [1,2,3,4]` vaut `[5,1,2,3,4]`.
- `liste1 + liste2` concatène les deux listes. Exemple : avec `liste1 = [4,5,6]` et `liste2 = [7,8,9]` alors `liste1 + liste2` vaut `[4,5,6,7,8,9]`.

**Exemple de construction.** Voici comment construire la liste qui contient les premiers carrés :

```
liste_carres = [] # On part d'une liste vide
for i in range(10):
 liste_carres.append(i**2) # On ajoute un carré
```

À la fin `liste_carres` vaut :

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

### Parcourir une liste

- `len(liste)` renvoie la longueur de la liste. Exemple : `len([5,4,3,2,1])` renvoie 5.
- Parcourir simplement une liste (et ici afficher chaque élément) :

```
for element in liste:
 print(element)
```

- Parcourir une liste à l'aide du rang.

```
n = len(liste)
for i in range(n):
 print(i, liste[i])
```

### Copier une liste

```
new_list = list(liste)
```

## 6. Listes II

### Mathématiques

- `max(liste)` renvoie le plus grand élément. Exemple : `max([10, 16, 13, 14])` renvoie 16.
- `min(liste)` renvoie le plus petit élément. Exemple : `min([10, 16, 13, 14])` renvoie 10.
- `sum(liste)` renvoie la somme de tous les éléments. Exemple : `sum([10, 16, 13, 14])` renvoie 53.

### Trancher des listes

- `liste[a:b]` renvoie la sous-liste des éléments du rang  $a$  au rang  $b - 1$ .
- `liste[a:]` renvoie la liste des éléments du rang  $a$  jusqu'à la fin.
- `liste[:b]` renvoie la liste des éléments du début jusqu'au rang  $b - 1$ .

| Lettre | "A" | "B" | "C" | "D" | "E" | "F" | "G" |
|--------|-----|-----|-----|-----|-----|-----|-----|
| Rang   | 0   | 1   | 2   | 3   | 4   | 5   | 6   |

Par exemple si `liste = ["A", "B", "C", "D", "E", "F", "G"]` alors :

- `liste[1:4]` renvoie `["B", "C", "D"]`.
- `liste[:2]` c'est comme `liste[0:2]` et renvoie `["A", "B"]`.
- `liste[4:]` renvoie `["E", "F", "G"]`. C'est la même chose que `liste[4:n]` où `n = len(liste)`.

### Trouver le rang d'un élément

- `liste.index(element)` renvoie la première position à laquelle l'élément a été trouvé. Exemple : avec `liste = [12, 30, 5, 9, 5, 21]`, `liste.index(5)` renvoie 2.
- Si on souhaite juste savoir si un élément appartient à une liste, alors l'instruction :

```
element in liste
```

renvoie `True` ou `False`. Exemple : avec `liste = [12, 30, 5, 9, 5, 21]`, «`9 in liste`» est vrai, alors que «`8 in liste`» est faux.

### Ordonner

- `sorted(liste)` renvoie la liste ordonnée des éléments.  
Exemple : `sorted([13, 11, 7, 4, 6, 8, 12, 6])` renvoie la liste `[4, 6, 6, 7, 8, 11, 12, 13]`.
- `liste.sort()` ne renvoie rien mais par contre la liste `liste` est maintenant ordonnée.

### Inverser une liste

Voici trois méthodes :

- `liste.reverse()` modifie la liste sur place ;
- `list(reversed(liste))` renvoie une nouvelle liste ;
- `liste[::-1]` renvoie une nouvelle liste.

### Supprimer un élément

Trois méthodes.



- `liste.remove(element)` supprime la première occurrence trouvée.  
Exemple : `liste = [2,5,3,8,5]`, la commande `liste.remove(5)` modifie la liste qui maintenant vaut `[2,3,8,5]` (le premier 5 a disparu).
- `del liste[i]` supprime l'élément de rang  $i$  (la liste est modifiée).
- `element = liste.pop()` supprime le dernier élément de la liste et le renvoie. C'est l'opération « dépiler ».

### Liste par compréhension

- Partons d'une liste, par exemple `maliste = [1,2,3,4,5,6,7,6,5,4,3,2,1]`.
- `liste_doubles = [ 2*x for x in maliste ]` renvoie une liste qui contient les doubles des éléments de la liste `maliste`. C'est donc la liste `[2,4,6,8,...]`.
- `liste_carres = [ x**2 for x in maliste ]` renvoie la liste des carrés de éléments de la liste `maliste`. C'est donc la liste `[1,4,9,16,...]`.
- `liste_partielle = [x for x in maliste if x > 2]` extrait la liste composée des seuls éléments strictement supérieurs à 2. C'est donc la liste `[3,4,5,6,7,6,5,4,3]`.

### Liste de listes

Exemple :

```
tableau = [[2,14,5], [3,5,7], [15,19,4], [8,6,5]]
```

correspond au tableau :

|            |       |            |       |       |
|------------|-------|------------|-------|-------|
|            |       | indice $j$ |       |       |
|            |       | →          |       |       |
|            |       | $j=0$      | $j=1$ | $j=2$ |
| indice $i$ | $i=0$ | 2          | 14    | 5     |
|            | $i=1$ | 3          | 5     | 7     |
|            | $i=2$ | 15         | 19    | 4     |
|            | $i=3$ | 8          | 6     | 5     |

Alors `tableau[i]` renvoie la sous-liste de rang  $i$ , et `tableau[i][j]` renvoie l'élément situé dans la sous-liste de rang  $i$ , au rang  $j$  de cette sous-liste. Par exemple :

- `tableau[0]` renvoie la sous-liste `[2,14,5]`.
- `tableau[1]` renvoie la sous-liste `[3,5,7]`.
- `tableau[0][0]` renvoie l'entier 2.
- `tableau[0][1]` renvoie l'entier 14.
- `tableau[2][1]` renvoie l'entier 19.

Un tableau de  $n$  lignes et  $p$  colonnes.

- `tableau = [[0 for j in range(p)] for i in range(n)]` initialise un tableau et le remplit de 0.
- `tableau[i][j] = 1` modifie une valeur du tableau (celle à l'emplacement  $(i,j)$ ).

## 7. Dictionnaire

Un dictionnaire est un peu comme une liste, mais les éléments ne sont pas indexés par des entiers mais par une « clé ». Un **dictionnaire** est donc un ensemble de couples clé/valeur : à une **clé** est associée une **valeur**.

### Exemple : un dictionnaire identifiant/mot de passe.

- Voici l'exemple d'un dictionnaire dico qui stocke des identifiants et des mots de passe :  

```
dico = {'jean':'rev1789', 'adele':'azerty', 'jasmine':'c3por2d2'}
```
- Par exemple 'adele' a pour mot de passe 'azerty'. On obtient le mot de passe comme on accéderait à un élément d'une liste par l'instruction :

```
dico['adele'] qui vaut 'azerty'.
```

- Pour ajouter une entrée on écrit :

```
dico['lola'] = 'abcdef'
```

- Pour modifier une entrée :

```
dico['adele'] = 'vwxyz'
```

- Maintenant la commande `print(dico)` affiche :

```
{'jean':'rev1789', 'adele':'vwxyz', 'jasmine':'c3por2d2', 'lola':'abcdef'}
```

- Le parcours d'un dictionnaire se fait par une boucle « pour ». Par exemple, la boucle suivante affiche l'identifiant et le login de tous les éléments du dictionnaire :

```
for prenom in dico:
 print(prenom + " a pour mot de passe " + dico[prenom])
```

- Attention : il n'y a pas d'ordre dans un dictionnaire. Tu ne contrôles pas dans quel ordre les éléments sont traités.

### Commandes principales.

- Définir un dictionnaire `dico = {cle1:valeur1, cle2:valeur2,...}`
- Récupérer une valeur : `dico[cle]`
- Ajouter une valeur : `dico[new_cle] = valeur`
- Modifier une valeur : `dico[cle] = new_valeur`
- Taille du dictionnaire : `len(dico)`
- Parcourir un dictionnaire : `for cle in dico:` et dans la boucle on accède aux valeurs par `dico[cle]`
- Tester si une clé existe : `if cle in dico:`
- Dictionnaire vide : `dico = {}`, utile pour initialiser un dictionnaire dans le but de le remplir ensuite.

Des commandes un peu moins utiles :

- Liste des clés : `dico.keys()`
- Liste des valeurs : `dico.values()`
- On peut récupérer les clés et les valeurs pour les utiliser dans une boucle :

```
for cle,valeur in dico.items():
 print("Clé :", cle, " Valeur :", valeur)
```

## 8. Entrée/sortie

### Affichage

- `print(chaine1, chaine2, chaine3, ...)` affiche des chaînes ou des objets. Exemple : `print("Valeur =", 14)` affiche `Valeur = 14`. Exemple : `print("Ligne 1 \n Ligne 2")` affiche sur deux lignes.
- **Séparateur.** `print(..., sep="...")` change le séparateur (par défaut le séparateur est le caractère espace). Exemple : `print("Bob", 17, 13, 16, sep="; ")` affiche `Bob; 17; 13; 16`.
- **Fin de ligne.** `print(..., end="...")` change le caractère placé à la fin (par défaut c'est le saut de ligne `\n`). Exemple `print(17, end="")` puis `print(89)` affiche `1789` sur une seule ligne.

### Entrée clavier

`input()` met le programme en pause et attend de l'utilisateur un message au clavier (qu'il termine en appuyant sur la touche « Entrée »). Le message est une chaîne de caractères.

Voici un petit programme qui demande le prénom et l'âge de l'utilisateur et affiche un message du style « Bonjour Kevin » puis « Tu es mineur/majeur » selon l'âge.

```
prenom = input("Comment t'appelles-tu ? ")
print("Bonjour", prenom)
```

```
age_chaine = input("Quel âge as-tu ? ")
age = int(age_chaine)
```

```
if age >= 18:
 print("Tu es majeur !")
else:
 print("Tu es mineur !")
```

## 9. Fichiers

### Commande

- `fic = open("mon_fichier.txt", "r")` ouverture en lecture ("`r`" = *read*).
- `fic = open("mon_fichier.txt", "w")` ouverture en écriture ("`w`" = *write*). Le fichier est créé s'il n'existe pas, s'il existait le contenu précédent est d'abord effacé.
- `fic = open("mon_fichier.txt", "a")` ouverture en écriture, les données seront écrites à la fin des données actuelles ("`a`" = *append*).
- `fic.write("une ligne")` écriture dans le fichier.
- `fic.read()` lit tout le fichier (voir plus bas pour autre méthode).
- `fic.readlines()` lit toutes les lignes (voir plus bas pour autre méthode).
- `fic.close()` fermeture du fichier.

### Écrire des lignes dans un fichier

```
fic = open("mon_fichier.txt", "w")
```

```
fic.write("Bonjour le monde\n")
```

```
ligne = "Coucou\n"
fic.write(ligne)
```

```
fic.close()
```

**Lire les lignes d'un fichier**

```

fic = open("mon_fichier.txt","r")

for ligne in fic:
 print(ligne)

fic.close()

```

*Utile.* La commande `ligne.strip()` renvoie la chaîne de caractères de la ligne sans les espaces de début et de fin, ni le saut de ligne.

**Lire un fichier (méthode officielle)**

```

with open("mon_fichier.txt","r") as fic:
 for ligne in fic:
 print(ligne)

```

## 10. Tortue

Le module `turtle` s'appelle par la commande :

```
from turtle import *
```

**Principales commandes**

- `forward(longueur)` avance de longueur pas
- `backward(longueur)` recule
- `right(angle)` tourne vers la droite selon l'angle donné en degrés
- `left(angle)` tourne vers la gauche
- `setheading(direction)` s'oriente dans une direction (0 = droite, 90 = haut, -90 = bas, 180 = gauche)
- `goto(x,y)` se déplace jusqu'au point (x,y)
- `setx(newx)` change la valeur de l'abscisse (déplacement horizontal)
- `sety(newy)` change la valeur de l'ordonnée (déplacement vertical)
- `down()` abaisse le stylo
- `up()` relève le stylo
- `width(epaisseur)` change l'épaisseur du trait
- `color(couleur)` change la couleur du trait : "red", "green", "blue", "orange", "purple",...
- `position()` renvoie la position (x,y) de la tortue
- `heading()` renvoie la direction angle vers laquelle pointe la tortue
- `towards(x,y)` renvoie l'angle entre l'horizontale et le segment commençant à la tortue et finissant au point (x,y)
- `speed("fastest")` vitesse maximale de déplacement
- `hideturtle()` cache le curseur de la tortue
- `showturtle()` affiche le curseur de la tortue
- `exitonclick()` termine le programme dès que l'on clique

**Plusieurs tortues**

Voici un exemple de programme avec deux tortues.

```

tortue1 = Turtle() # Avec un 'T' majuscule !
tortue2 = Turtle()

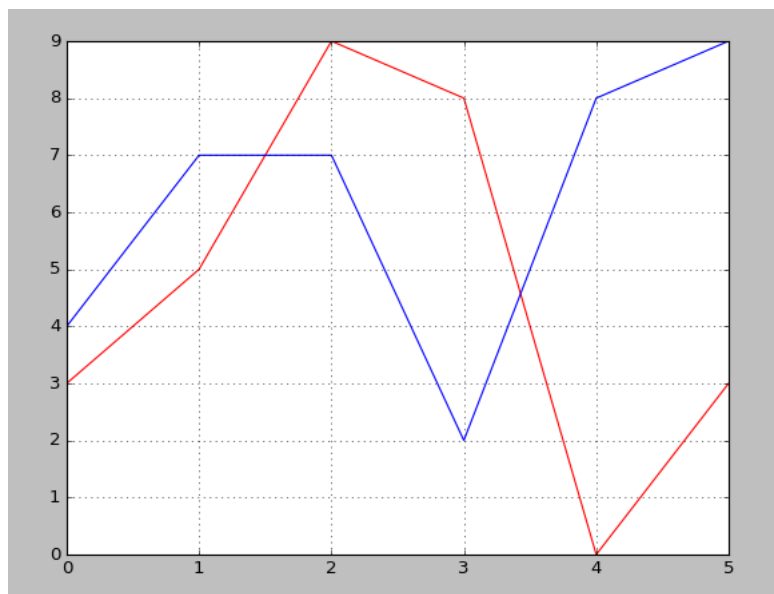
```

```
tortue1.color('red')
tortue2.color('blue')

tortue1.forward(100)
tortue2.left(90)
tortue2.forward(100)
```

## 11. Matplotlib

Avec le module matplotlib il est très facile de tracer une liste. Voici un exemple.



```
import matplotlib.pyplot as plt
```

```
liste1 = [3,5,9,8,0,3]
liste2 = [4,7,7,2,8,9]
```

```
plt.plot(liste1,color="red")
plt.plot(liste2,color="blue")
plt.grid()
plt.show()
```

### Principales fonctions.

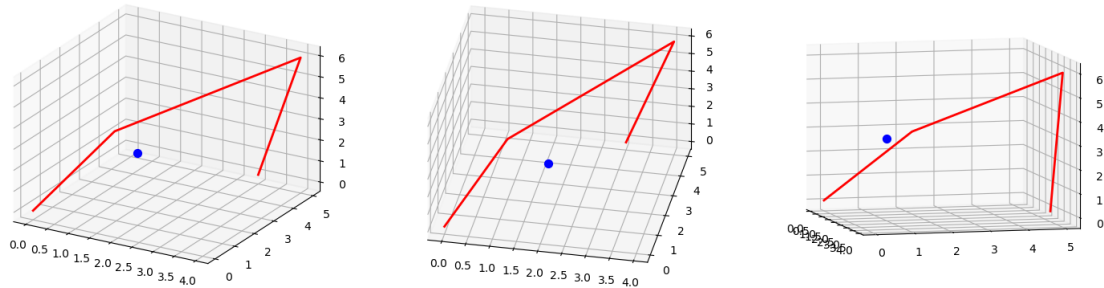
- `plt.plot(liste)` trace les points d'une liste (sous la forme  $(i, \ell_i)$ ) et les joint.
- `plt.plot(listex, listey)` trace les points d'une liste (sous la forme  $(x_i, y_i)$ ) où  $x_i$  parcourt la première liste et  $y_i$  la seconde).
- `plt.scatter(x, y, color='red', s=100)` affiche un point en  $(x, y)$  (d'une grosseur  $s$ ).
- `plt.grid()` trace une grille.
- `plt.show()` affiche tout.
- `plt.close()` termine le tracé.
- `plt.xlim(xmin, xmax)` définit l'intervalle des  $x$ .
- `plt.ylim(ymin, ymax)` définit l'intervalle des  $y$ .
- `plt.axis('equal')` impose un repère orthonormé.

## 12. Matplotlib 3D

Avec le module `matplotlib` il est aussi assez facile de tracer une représentation des objets dans l'espace. Le principe est similaire à l'affichage dans le plan, sauf bien sûr qu'il faut préciser trois coordonnées  $x, y, z$  pour déterminer un point de l'espace.

Voici un code très simple qui affiche :

- un point bleu de coordonnées (2, 1, 3),
- des segments rouges qui relient les points de la liste (0, 0, 0), (1, 2, 3), (4, 5, 6), (3, 5, 0).



Une fenêtre s'affiche dans laquelle sont dessinés le point et les segments ainsi que les plans quadrillés de coordonnées. L'image est dynamique : à l'aide de la souris tu peux faire tourner le dessin afin de changer de point de vue.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

Initialisation
fig = plt.figure()
ax = fig.gca(projection='3d', proj_type = 'ortho')

Affichage d'un point
x,y,z = (2,1,3)
ax.scatter(x,y,z,color='blue',s=50)

Segments reliant des points
points = [(0,0,0),(1,2,3),(4,5,6),(3,5,0)]
liste_x = [x for x,y,z in points]
liste_y = [y for x,y,z in points]
liste_z = [z for x,y,z in points]

ax.plot(liste_x,liste_y,liste_z,color='red',linewidth=2)

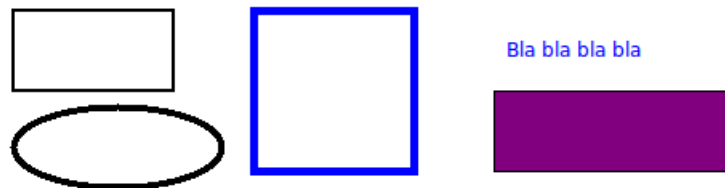
Affichage
plt.show()
```

*Avertissement.* Pour afficher des segments la commande `plot` n'est pas très naturelle (mais c'était déjà le cas dans le plan). Par exemple pour relier le point (1, 2, 3) au point (4, 5, 6) on donne d'abord la liste des  $x$ , puis la liste des  $y$ , puis la liste des  $z$  :

```
plot([1,4],[2,5],[3,6])
```

## 13. Tkinter

Pour afficher ceci :



le code est :

```
Module tkinter
from tkinter import *

Fenêtre tkinter
root = Tk()

canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(fill="both", expand=True)

Un rectangle
canvas.create_rectangle(50,50,150,100,width=2)

Un rectangle à gros bords bleus
canvas.create_rectangle(200,50,300,150,width=5,outline="blue")

Un rectangle rempli de violet
canvas.create_rectangle(350,100,500,150,fill="purple")

Un ovale
canvas.create_oval(50,110,180,160,width=4)

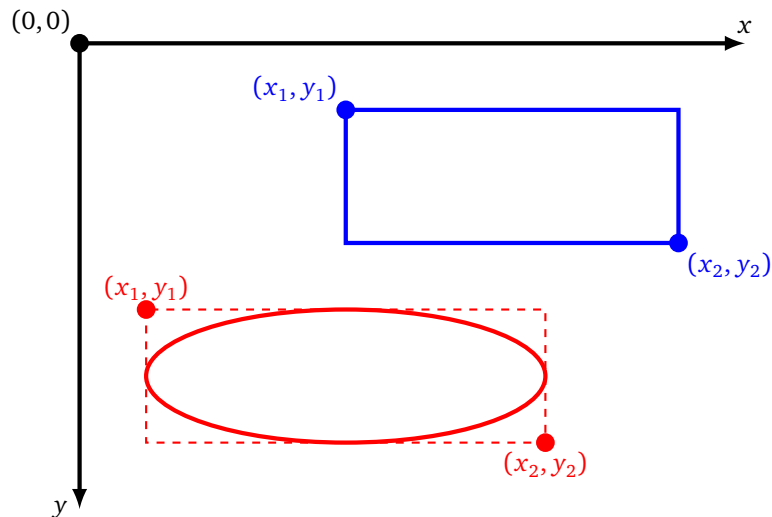
Du texte
canvas.create_text(400,75,text="Bla bla bla bla",fill="blue")

Ouverture de la fenêtre
root.mainloop()
```

Quelques explications :

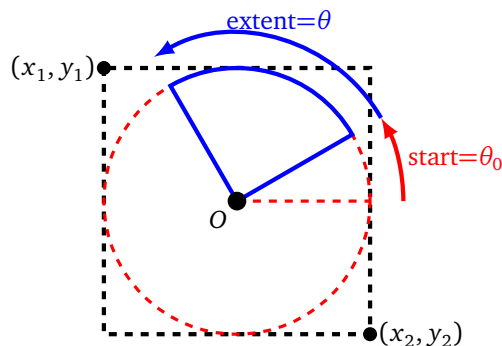
- Le module tkinter nous permet de définir des variables `root` et `canvas` qui définissent une fenêtre graphique (ici de largeur 800 et de hauteur 600 pixels). On décrit ensuite tout ce que l'on veut ajouter dans la fenêtre. Et enfin, la fenêtre est affichée par la commande `root.mainloop()` (tout à la fin).
- Attention ! Le repère graphique de la fenêtre a son axe des ordonnées dirigé vers le bas. L'origine (0,0) est le coin en haut à gauche (voir la figure ci-dessous).
- Commande pour tracer un rectangle : `create_rectangle(x1,y1,x2,y2)` ; il suffit de préciser les coordonnées  $(x_1, y_1)$  et  $(x_2, y_2)$  de deux sommets opposés. L'option `width` ajuste l'épaisseur du trait, `outline` définit la couleur de ce trait et `fill` définit la couleur de remplissage.

- Une ellipse est tracée par la commande `create_oval(x1,y1,x2,y2)`, où  $(x_1, y_1)$ ,  $(x_2, y_2)$  sont les coordonnées de deux sommets opposés d'un rectangle encadrant l'ellipse voulue (voir la figure). On obtient un cercle lorsque le rectangle correspondant est un carré.
- Le texte est affiché par la commande `canvas.create_text()` en précisant les coordonnées  $(x, y)$  du point à partir duquel on souhaite afficher le texte.



**Portion de cercle.** La fonction `create_arc()` n'est pas très intuitive. Il faut penser que l'on dessine un cercle, en précisant les coordonnées de deux sommets opposés d'un carré qui l'entoure, puis en précisant l'angle de début et l'angle du secteur (en degrés).

`canvas.create_arc(x1,y1,x2,y2,start=debut_angle,extent=mon_angle)`



L'option `style=PIESLICE` affiche un secteur au lieu d'un arc.



## Notes et références

Tu trouveras ici des commentaires et des indications de lectures sur chacune des activités.

### Ressources générales

- *Python au lycée – tome 1*. Indispensable !
  - À télécharger ici : [exo7.emath.fr](http://exo7.emath.fr)
  - Version papier en vente à prix coûtant : [amazon.fr](http://amazon.fr)
  - Tous les codes Python ainsi que les fichiers sources : [github.com/exo7math/python1-exo7](https://github.com/exo7math/python1-exo7)
  - Les vidéos d'introduction à Python : [youtube.com/PythonAuLycée](https://youtube.com/PythonAuLycée)
- *Apprendre à programmer avec Python 3* de Gérard Swinnen, éditions Eyrolles. Le livre est disponible gratuitement en téléchargement, selon la licence *Creative Commons BY-NC-SA* :  
[inforef.be/swi/python.htm](http://inforef.be/swi/python.htm)
- La documentation officielle de Python :  
[docs.python.org/fr/3/](https://docs.python.org/fr/3/)

### 1. Suites arithmétiques – Suites géométriques

Ces deux types de suites sont idéales pour réviser les différentes formules permettant de calculer les termes soit directement, soit par récurrence. On peut regretter que Python n'effectue pas des calculs exacts même pour les opérations élémentaires. Par exemple les fractions sont transformés en nombres flottants :

- $1/3$  renvoie  $0.33333...$
- $1/10 + 1/5$  renvoie  $0.30000000000000004$  et n'est pas égal à  $3/10$ .

### 2. Nombres complexes I

C'est une activité assez simple pour se remettre à Python en terminale. C'est très agréable que Python calcule directement avec les nombres complexes avec encore le bémol que les calculs s'effectuent en nombres flottants. L'activité incontournable est de résoudre les équations du second degré. On insistera sur la programmation sous forme de fonctions. Attention une fonction n'est pas là pour afficher des résultats, mais une fonction renvoie des valeurs. C'est aussi l'occasion d'insister sur l'aspect géométrique des nombres complexes.

### 3. Nombres complexes II

Le module `cmath` permet d'aller plus loin avec les nombres complexes et la forme `module/argument`. On peut se contenter d'utiliser les fonctions prédéfinies mais ici on propose en plus de reprogrammer ces fonctions. Les coordonnées polaires sont importantes au delà des nombres complexes que ce soit en mathématique ou en physique. Encore une fois on insiste sur la visualisation.

Notez l'astuce de Gauss (1777-1855) pour réduire le produit de deux nombres complexes avec seulement trois multiplications réelles. C'est une astuce similaire à la base de la multiplication rapide de Karatsuba (en 1960). La notion de complexité est abordée dans un autre chapitre.

### 4. Dérivée – Zéros de fonctions

Les notions de dérivée et de tangente sont essentielles pour l'étude de fonction. À l'aide de Python on peut trouver des valeurs approchées de limites et donc des dérivées. Savoir programmer le tracé du graphe d'une fonction permet de mieux appréhender la calcul infinitésimal et la notion de « epsilon ».

Deux activités sont fondamentales : la dichotomie et la méthode de Newton. La dichotomie est à la base du concept « diviser pour régner » et on retrouvera ce concept lors de la recherche dans une liste (chapitre « Le mot le plus long »). La méthode de Newton, qui utilise la notion de tangente, permet de comprendre qu'un bon algorithme est plus important qu'un ordinateur puissant !

### 5. Exponentielle

Il est important d'appréhender la croissance de l'exponentielle (c'est toujours plus que l'on « croît »). L'exponentielle est essentielle pour la modélisation de nombreux phénomènes physiques, par contre c'est la bête noire des informaticiens car tout phénomène exponentiel déborde rapidement les capacités d'un ordinateur (voir le chapitre « Tri - Complexité »).

### 6. Logarithme

Le logarithme a autant d'importance que l'exponentielle. La difficulté supplémentaire c'est qu'il y a plusieurs bases et plusieurs notations pour le logarithme. Le mieux est quand même de commencer par le logarithme décimal, vu comme l'exposant d'une puissance de 10. Mais bien sûr le logarithme népérien est préféré en mathématiques et le logarithme en base 2 est préféré en informatique. L'histoire du calcul des logarithmes à travers les âges est illustrée par quelques algorithmes.

### 7. Intégrale

Il s'agit d'une activité très facile du point vue informatique. Le calcul approché d'intégrales est indispensable lorsque qu'on ne peut pas calculer de primitives pour une fonction, ce qui est le cas dans la « vraie vie ». Le calcul approché est de nouveau l'occasion de comparer l'efficacité des différents algorithmes.

## 8. Programmation objet

La programmation objet est une façon différente de concevoir ses programmes. Si on veut mieux comprendre Python il faut comprendre ce concept. Mais à part le chapitre suivant, toutes les activités de ce livre peuvent se passer de programmation objet. L'avantage de la programmation objet, c'est qu'un l'objet contient à la fois ses variables et ses fonctions. L'exemple de plusieurs tortues devrait être éclairant : chaque tortue a sa position, sa couleur, chaque tortue peut avancer indépendamment des autres tortues. Par contre les premiers pas en programmation objet sont délicats et il faut beaucoup de lignes de code même pour un petit programme.

## 9. Mouvement de particules

C'est l'activité où la programmation objet prend tout son sens : chaque particule est un objet indépendant mais qui cependant interagit avec les autres particules. Pour ceux qui cherchent des idées en lien avec la programmation objet et la modélisation du monde qui nous entoure, le site :

[natureofcode.com](http://natureofcode.com)

est plein de projets, mais codés avec un autre langage. Le livre et sa version en ligne sont gratuits.

## 10. Algorithmes récursifs

Il s'agit d'un concept avancé de programmation qui demande pas mal de remue-méninges. La similarité entre le principe de récurrence et la récursivité, illustré par l'inusable exemple de la factorielle, permet de bien démarrer. Mais la récursivité permet d'écrire de façon simple des successions compliquées et imbriquées de solutions. Voir le déroulé complet d'appels récursifs reste fascinant ! C'est ce que l'on fait dans les activités graphiques à la fin. C'est sûrement un des chapitres qu'il vaut mieux étudier par petites touches en plusieurs fois. On retrouvera la récursivité dans d'autres chapitres.

## 11. Tri – Complexité

Deux notions très importantes en informatique. Tout d'abord la notion de tri, bien que le terme juste soit celui d'ordre. L'intérêt est immédiat : on n'imagine pas chercher un mot dans un dictionnaire dont les noms ne seraient pas classés par ordre alphabétique. Il existe de nombreux algorithmes de tri chacun apportant de petites ou de grandes améliorations. On présente ici seulement trois algorithmes simples, le meilleur algorithme nécessitant la récursivité.

La notion de complexité est très théorique même si on comprend bien la différence entre un algorithme lent ou rapide. On n'aborde pas ici la notion de complexité pour la mémoire. La notion de complexité est liée à la notion de limite : un algorithme en  $O(n^2)$  peut être plus rapide qu'un algorithme en  $O(n)$  pour de petites valeurs de  $n$ , mais pour  $n$  « assez grand » c'est l'algorithme en  $O(n)$  le plus rapide. C'est l'occasion de comparer les suites  $(\ln n)$ ,  $(n)$  et  $(e^n)$ .

## 12. Calculs en parallèle

Les processeurs sont de plus en plus puissants (voir la loi de Moore qui donne un bon exemple de phénomène exponentiel) mais pour aller encore plus vite il faut distribuer le travail : c'est le calcul parallèle. L'idée est simple mais la mise en œuvre est complexe, d'ailleurs avec Python il n'est pas facile d'effectuer des tâches en parallèle. Les activités proposées sont seulement des simulations de calcul parallèle.

## 13. Automates

Les automates sont une version simple du « jeu de la vie ». On peut étudier toutes les règles possibles et c'est l'occasion de revoir l'écriture binaire.

## 14. Cryptographie

Union parfaite des mathématiques et de l'informatique : la cryptographie ! Il faut tout d'abord comprendre qu'un message secret se transforme en une suite d'entiers et ensuite que le chiffrement et le déchiffrement sont des opérations mathématiques. L'aspect le plus amusant c'est l'attaque d'un message secret : les ordinateurs sont capables de rechercher des milliers de combinaisons en quelques secondes. Par contre c'est une grave erreur de croire que c'est le nombre de combinaisons qui fait la solidité d'un système de chiffrement. C'est l'erreur faite par les Allemands lors de la seconde guerre mondiale avec la machine *Enigma*. Grâce à Turing les Alliés ont pu décrypter les messages des Allemands. L'histoire de la cryptographie est racontée dans le livre *Histoire des codes secrets* de Simon Singh (Le livre de Poche).

## 15. Le compte est bon

« Le compte est bon » est exactement le genre de problème difficile à résoudre pour un humain mais simple pour un ordinateur. Le nombre de combinaisons à étudier n'est pas très élevé et une recherche exhaustive bien menée suffit à trouver une solution. Ce qui complique la mise en œuvre c'est qu'il ne faut pas se contenter des solutions séquentielles (voir l'activité) ce qui conduit à un algorithme récursif.

## 16. Le mot le plus long

Encore un problème qui est de petite taille pour un ordinateur mais qui fait réfléchir les humains. Par contre pour le rendre simple à un ordinateur il faut prendre le problème par le bon bout. Un humain prend les lettres et à partir des lettres cherche des mots français. Ce n'est pas la bonne approche pour un ordinateur car il y a beaucoup de tirages possibles et beaucoup de combinaisons pour chaque tirage. Par contre des mots français il y en a environ 100 000, ce qui est peu pour un ordinateur. Donc avec la méthode d'indexation et un bon algorithme de recherche dans une liste triée c'est très facile.

## 17. Images et matrices

Cette activité n'est pas très difficile : pour le début une matrice est juste utilisée comme un tableau de nombres. L'application de la convolution sur les images est assez impressionnante et utilisée telle quelle dans les logiciels de retouche d'images. Pour déformer les images on considère cette fois les matrices comme des applications du plan dans lui-même.

## 18. Ensemble de Mandelbrot

La fractale de Mandelbrot demande un peu d'effort de la part du programmeur et beaucoup de calculs pour l'ordinateur. Par contre tout ce travail vaut le coup et vous plonge dans le monde infini des fractales. On peut programmer cette activité sans connaître les nombres complexes. Si on veut vraiment dessiner beaucoup de fractales avec Python il faut chercher des méthodes pour accélérer les calculs d'un facteur 100 (voir cython, numba, numpy, pycuda).

## 19. Images 3D

Pour visualiser des objets en dimension 3, `matplotlib` est appréciable car on peut faire tourner les objets dans l'espace. Pour bien comprendre que l'on ne dessine que des images du plan il faut étudier les différentes projections possibles. Une autre écriture possible des perspectives serait d'utiliser les matrices. Le calcul vectoriel est très utile pour la visualisation en dimension 3 car les objets compliqués sont représentés par une multitude de triangles de l'espace où chaque triangle est traité indépendamment. Par exemple pour connaître l'éclairage d'un triangle on effectue le produit scalaire entre un rayon lumineux et un vecteur normal au triangle.

## 20. Sudoku

Encore un problème qui fait transpirer les humains plus que les ordinateurs. Mais ici il faut en plus se creuser la tête pour programmer la résolution. Le « retour en arrière » (*backtracking* en anglais) est une méthode générale qui permet de résoudre de nombreux problèmes (ici le problème des 8 reines et le sudoku).

## 21. Fractale de Lyapunov

Une nouvelle série de fractales dans la continuité de Mandelbrot. Cependant il faut connaître un peu plus de mathématiques (les logarithmes) et les calculs sont encore plus longs.

## 22. Big data I

La statistique de base (moyenne, écart-type, régression linéaire, distribution de Gauss...) fournit des outils puissants pour le traitement des données même de grande taille. Notez que dans l'activité sur la classification bayésienne naïve les formules sont présentées sous une forme simplifiée.

Les identités fictives utilisées dans les activités ont été créées à l'aide de l'outil rigolo :

[fr.fakenamegenerator.com](http://fr.fakenamegenerator.com)

## 23. Big data II

Lorsque l'on parle de « big data » on parle de données dépassant le téraoctet. Il est alors trop difficile ou trop long de traiter les données une par une. Les activités présentées sont des méthodes récentes adaptées à ces volumes d'informations. Le test probabiliste d'un parenthésage correct est basé sur le cours « Algorithmes de flux de données » de Claire Mathieu au Collège de France. Le perceptron est avant tout lié à l'intelligence artificielle et plus spécifiquement aux réseaux de neurones.

## Remerciements

Je remercie François Recher pour sa relecture attentive, ses commentaires et encouragements. Je remercie Michel Bodin, Pascal Ortiz et Pascal Veron pour leur relecture.

Vous pouvez récupérer l'intégralité des codes Python des activités ainsi que tous les fichiers sources sur la page *GitHub* d'Exo7 : « [GitHub : Python au lycée](#) ».

Les vidéos du tome 1 sont disponibles sur la chaîne *Youtube* : « [Youtube : Python au lycée](#) ».



Ce livre est diffusé sous la licence *Creative Commons – BY-NC-SA – 4.0 FR*.  
Sur le site Exo7 vous pouvez télécharger gratuitement le livre en couleur.

- abs, 9
- algorithme récursif, 79, 138
- angle, 182
- arbre, 85, 138
- automate, 123
  
- barycentre, 230
- big data*, 212, 224
- bin, 148
- binaire, 148
- break, 246
  
- calcul parallèle, 107
- chaîne de caractères
  - eval, 12
  - replace, 12
  - split, 12
- class, 58
- clé, 144
- close, 143
- coefficients du binôme de Newton, 83
- complexité, 97
- conjugate, 9
- continue, 246
- convolution, 150
- couleurs, 238
- cryptographie, 129
  - attaque, 131
  - chiffre de César, 129
  - chiffrement de Vigenère, 134
  - chiffrement par substitution, 131
  
- dérivée, 22
- dichotomie, 27, 46, 143
- dictionnaire, 144
- double affectation, 100
  
- elif, 245
  
- équation du second degré, 12
- eval, 12
- exp, 32
- exponentielle, 31, 97
- exponentielle complexe, 17
  
- factorielle, 79, 89
- fichier, 143
- fonction
  - return, 249
- fonction lambda, 22
- fractale
  - courbe de Hilbert, 95
  - ensemble de Mandelbrot, 162
  - ensembles de Julia, 167
  - flocon de Koch, 92
  - fractale de Lyapunov, 202
  - triangle de Sierpinski, 94
  
- grand O, 97
  
- imag, 9
- image, 150, 169
- index, 130
- infini, 116
- intégrale, 52
- isinstance, 88
- items, 145
  
- keys, 145, 264
  
- lambda, 22
- latitude/longitude, 179
- log, 38
- logarithme, 37, 97, 207
  - décimal, 37
  - entier, 46
  - népérien, 42



matrice, [61](#), [62](#), [150](#)  
médiane, [8](#)  
méthode, [58](#)  
méthode de Newton, [29](#)  
module  
    `cmath`, [15](#)  
    `matplotlib`, [10](#), [169](#)  
    `tkinter`, [73](#)  
neurone, [236](#)  
nombre complexe, [9](#), [15](#), [162](#)  
    argument, [15](#)  
    `cmath`, [15](#)  
    module, [15](#)  
objet, [58](#)  
open, [143](#)  
  
particule, [70](#)  
pass, [247](#)  
permutation, [90](#)  
perspective, [176](#)  
*pgm*, [153](#)  
phase, [15](#)  
pile, [189](#)  
polar, [16](#)  
primitive, [52](#)  
programmation objet, [58](#)  
    classe, [58](#)  
    convivialité, [60](#)  
    encapsulation, [68](#)  
    hérité, [66](#)  
    méthode, [58](#)  
puissance, [32](#)  
  
real, [9](#)  
  
rect, [16](#), [257](#)  
récursivité, [79](#)  
replace, [12](#)  
retour en arrière, [189](#)  
return, [249](#)  
*rgb*, [238](#)  
  
sort/sorted, [100](#)  
split, [12](#)  
strip, [144](#), [260](#), [266](#)  
sudoku, [189](#)  
suite, [97](#)  
    arithmétique, [2](#)  
    de Fibonacci, [83](#)  
    géométrique, [4](#)  
    logistique, [202](#)  
  
table de hachage, [113](#)  
tangente, [22](#)  
tortue, [5](#), [65](#), [67](#), [92](#)  
tri, [97](#)  
    à bulles, [103](#)  
    fusion, [103](#)  
    insertion, [101](#)  
    sélection, [100](#)  
triangle de Pascal, [84](#)  
  
values, [145](#), [264](#)  
vecteur, [58](#), [181](#)  
    produit scalaire, [181](#)  
    produit vectoriel, [181](#)  
  
write, [143](#)  
  
zéros, [22](#)