# Travaux Pratiques n°1
# Introduction à Python

## I. L'environnement

### A. Le système d'exploitation : Windows

▷ Se connecter sur l'ordinateur à l'aide de son identifiant `Magret` et de son mot de passe.

On accède ainsi au système d'exploitation de l'ordinateur, il s'agit ici de `Windows`.

▷ Cliquer sur `Ordinateur`, puis sur le disque `P:\Travail`. Créer un répertoire `Python`, le faire glisser vers le bureau tout en pressant la touche ALT pour créer un raccourci.

Il ne faut pas confondre le répertoire `Mes Documents` de l'ordinateur que vous utilisez actuellement avec celui de votre dossier personnel, qui doit être dans le disque dur du réseau (le disque `P`).

Créer dans son répertoire `Python` un répertoire pour cette première séance, `TP01` par exemple. Durant l'année un répertoire similaire sera créé pour chaque TP.

Les caractères spéciaux type `&`, `@` ou `#` ne sont pas autorisés dans les noms de fichiers. Les espaces sont déconseillés. Par contre les tirets (`-`) et tirets bas (underscore : _) sont autorisés.

### B. Utilisation du clavier

Pour bien utiliser son clavier, il est recommandé de placer les deux pouces sur la barre d'espace et les petits doigts sur les touches des bords droit et gauche du clavier, comme la touche Return (retour) ou les touches Shift. Les plus utiles de ces touches sont :

- CTRL, ALT, ALT GR : servent pour les raccourcis clavier, voir ci-dessous.
- Shift : la flèche montante de chaque côté des lettres sur le clavier, au dessus des CTRL : si elle est utilisée en combinaison avec une lettre, celle-ci est donnée en majuscule.
- Caps Lock (au dessus du Shift) permet de bloquer le Shift, donc toutes les lettres tapées seront obtenues en majuscule. Cette touche est déconseillée, car on tape rarement un mot entièrement en majuscule, on se contente généralement du Shift.
- Tab (au dessus du Caps Lock) : touche de *tabulation*. Très pratique, elle permet de faire un saut de plusieurs colonnes. Nous l'utiliserons en `Python` pour les indentations.
  Elle permet aussi, lorsque plusieurs champs sont à remplir dans une page web, de passer d'un champ au suivant. Utilisée en combinaison avec Shift, elle effectue l'opération inverse, c'est-à-dire qu'elle retourne au champ précédent.
- Echap ou Esc : touche d'échappement (Escape), permet parfois de sortir d'un programme, ou de retourner à la situation antérieure.

- Les touches Home (début), End (fin), Page Up et Page Down permettent de déplacer le curseur dans un fichier. Elles l'envoient respectivement au début du fichier, à la fin, à la page précédente, ou à la page suivante.
- Les touches Suppr ou Del effacent le caractère à droite du curseur, alors que la touche au-dessus de la touche Entrée efface le caractère à gauche du curseur.
- Pour les touches contenant plusieurs caractères, c'est le caractère du bas qui est obtenu, sauf si on les utilise en combinaison avec la touche Shift, auquel cas on obtient le caractère du haut, ou avec la touche ALT GR, auquel cas on obtient le caractère à droite de la touche (l'arobase @ est obtenu de cette façon).

Les systèmes d'exploitation proposent chacun une série de raccourcis clavier. Il s'agit de combinaisons de touches qui permettent d'accéder rapidement à certaines commandes. Ces commandes sont également accessibles grâce à la souris (avec un clic droit par exemple, ou en passant par la barre de menu en haut de la fenêtre), mais lorsqu'on travaille sur ordinateur on a souvent les mains sur le clavier, donc il est intéressant de les connaître. Voici la liste des plus utiles :

- CTRL-C pour copier : grâce à la souris on sélectionne une zone de texte, ou un ou plusieurs fichiers. Ils sont alors surlignés (grisés). L'utilisation de CTRL-C enregistre cette zone ou ces fichiers dans une mémoire volatile appelée *presse-papiers*.
- CTRL-X pour couper : cette combinaison supprime la zone sélectionnée et l'enregistre dans le presse-papiers. On peut alors la coller ailleurs.
- CTRL-V pour coller : cette combinaison recopie à la place du curseur le contenu du presse-papiers, c'est-à-dire la zone de texte qui a été copiée ou coupée.
- CTRL-S pour sauvegarder le fichier sur lequel on travaille. Il faut prendre l'habitude d'utiliser cette combinaison très souvent.
- CTRL-A sélectionne tout.
- CTRL-Z annule la dernière opération, par exemple la saisie de la dernière phrase.

Il existe aussi beaucoup d'autres raccourcis, en particulier les combinaisons avec la touche Windows (une petite fenêtre volante, à gauche entre CTRL et ALT), qui dépendent de la version de `Windows`.

### C. Idle

Pour écrire des programmes en Python et les exécuter on utilise un *environnement de développement intégré* (EDI).

Pour cette première séance nous utiliserons Idle. C'est un environnement minimal, écrit par le créateur de Python et livré avec toute version de Python.

▷ Lancer Idle. Pour ceci, aller dans le menu démarrer, taper `Idle` dans la barre de recherche. Il est possible de créer un raccourci sur le bureau.

La fenêtre Python Shell apparaît. C'est *l'interpréteur* (*shell* en anglais). On y tape des instructions en Python.

▷ **Exercice 1.** Taper et exécuter les expressions suivantes (presser la touche Entrée entre chaque expression) et répondre aux questions ci-dessous.

```
2+5   2*5   2**5   5/2   5.0/2   1/10000   2e5   1+3*10   (1+3)*10   3*10**2
```

La fenêtre doit ressembler à ceci :

```
>>> 2+5
7
>>> 2*5
10
...
```

Que représente l'opérateur `**` ?

Quel symbole représente le séparateur décimal ?

Comment est notée l'écriture scientifique d'un réel $a \times 10^n$ ?

Les priorités usuelles des opérateurs ($*$, $+$, $**$) sont-elles respectées ?

Taper maintenant les instructions suivantes :

```
23/5        23//5        23%5
```

Que représentent les opérateurs `//` et `%` ?

L'*interpréteur* permet de faire des petits calculs. Pour des tâches plus complexes, on écrit un programme dans l'*éditeur*.

On ouvre une nouvelle fenêtre depuis l'interpréteur grâce à la combinaison CTRL-N, ou en utilisant le menu `File - New window` ou `Fichier - Nouvelle fenêtre`. Il s'agit de *l'éditeur* et non plus de l'interpréteur.

C'est dans cet éditeur que l'on écrit les *programmes*, qui sont ensuite exécutés dans l'interpréteur.

▷ **Exercice 2.** Ouvrir une nouvelle fenêtre. Sauvegarder immédiatement le fichier (CTRL-S) dans un dossier correct et sous un nom correct (`Ex02.py` par exemple, mais l'extension `py` est automatiquement ajoutée). Taper les lignes suivantes :

```
print("Bonjour")
2*7
print(3*7)
4*7
a=5*7
print(a)
```

Sauvegarder le fichier (CTRL-S), l'exécuter grâce à la touche F5 et constater le résultat dans l'interpréteur.

Pourquoi 14 et 28 ne sont-ils pas affichés ?

Quitter `Idle` (CTRL-Q ou menu `File - Exit`). Le relancer, ouvrir le fichier de l'exercice précédent (CTRL-O), l'exécuter (F5) après avoir supprimé les instructions inutiles.

▷ **Exercice 3.** Ouvrir une nouvelle fenêtre (CTRL-N), la sauvegarder (CTRL-S) sous le nom `Ex03.py`. Saisir dans l'éditeur :

```
for i in range(1,7):
    print(i**2)
print("La valeur finale de i est",i)
```

Attention à respecter l'alignement : normalement, après les deux points, le curseur est automatiquement placé à la cinquième colonne. On dit que la ligne est *indentée*.

Sauver, exécuter. Expliquer. Que se passe-t-il si la troisième ligne est également indentée ?

Revenir à l'éditeur, fermer la fenêtre en cours (ALT-F4).

## II. Variables

Une *variable* est une zone de la mémoire de l'ordinateur qui

- porte un nom
- contient de l'information (une *valeur*).

Le nom est une chaîne de caractère arbitrairement longue, éventuellement avec des chiffres (mais pas en première position).

Attention : Les minuscules et les majuscules sont différenciées : `x1` est différent de `X1`.

La valeur peut être un entier, un réel (nous dirons un *flottant*), un caractère (`'a'` ou `"a"` par exemple), une chaîne de caractères (`'Bonjour, ça va ?'`, `"Oui, et toi ?"`), ou encore d'autres choses. Ce sont les *types* de variables.

On *affecte* la variable en attribuant une valeur à son nom. En `Python` on utilise l'instruction `x=5`, qui se lit *x reçoit* 5. En algorithmique on écrit plutôt $x \leftarrow 5$. En `Pascal` on utilise `x:=5`.

On peut ensuite utiliser cette variable, par exemple l'instruction `x+3` renverra `8`. On peut réaffecter la variable à tout moment : `x=14`. La valeur précédente sera oubliée.

On peut aussi *incrémenter* une variable grâce à l'instruction `x=x+1`.

▷ **Exercice 4.** Dans l'*interpréteur* taper les instructions suivantes (en pressant la touche Entrée après chacune d'entre elles) :

```
x=10        y=x        x=15
```

Que valent finalement `x` et `y` ? `x=` `y=`

Vérifier (en tapant `x` puis `y`).

Taper ensuite :

```
x,y=8,5        x        y        x*y
```

Il s'agit d'une *affectation multiple*.

▷ **Exercice 5.** Créer une nouvelle fenêtre (CTRL-N) de l'éditeur, taper les instructions suivantes **sans exécuter le programme** :

```
x=10
y=15
z=x+y
x=y
y=z
print(x+y+z)
```

Que va-t-il se passer à l'exécution ?

Vérifier en exécutant.

▷ **Exercice 6.** Inversion des valeurs de deux variables.

Après chacune de trois séquences ci-dessous, quelles valeurs ont les variables $x$ et $y$ ? Les deviner, puis vérifier.

| | | | |
|---|---|---|---|
| a. | `x=42 y=10 x=y y=x` | `x=` | `y=` |
| b. | `x=42 y=10 z=x x=y y=z` | `x=` | `y=` |
| c. | `x=42 y=10 x,y=y,x` | `x=` | `y=` |

▷ **Exercice 7.** Le type *tableau* ou *liste* (`list`)

Il s'agit de variables contenant plusieurs données ordonnées, chacune portant un numéro à partir de 0.
Exécuter dans l'interpréteur et comprendre les commandes suivantes :

`A=[6, 4, 1.2, 213, 0, 2/3, 1, 'Ab']` (Les espaces sont optionnels)

`A` `A[1]` `A[0]` `A[4]` `A[-1]` `len(A)` `A[1]=418` `A`

Que renvoie `A[k]` ?

Que renvoie `A[-1]` ?

Que signifie **len**`(A)` ?

▷ **Exercice 8.** Le type *chaîne de caractères* (`str`)

Exécuter maintenant :

`x='Bonjour'` `y='Adieu'` `x[1]` `y[2]` `x[7]` `y[-2]`

Puis :

`z=x+y` `z` `t=x+' et '+y` `t` `print(t)` `print(4*x)`

L'opérateur `+` sur les chaînes de caractères réalise la *concaténation*.

## III. Boucles, tests

### A. Observation

▷ **Exercice 9.** Boucles `for` et `while`

Dans l'éditeur, taper puis exécuter les lignes suivantes :

```
for i in range(10,15):
    print("Bonjour",i)
```

Quelles valeurs prend successivement `i` ?

Saisir puis exécuter maintenant : (*while* signifie «tant que»)

```
i=10
while i<15:
    print("Bonjour",i)
    i=i+1
```

Le résultat est-il identique ?

▷ **Exercice 10.** Instruction conditionnelle `if`

Dans l'éditeur, taper les deux programmes suivants.

```
if 10**2<99:
    print('Paf')
if 5**2>20:
    print('Pif')
if 10**2<99 or 5**2>20:
    print('Pof')
```

```
if 10**2<99:
    print('Paf')
    if 5**2>20:
        print('Pif')
if 10**2<99 or 5**2>20:
    print('Pof')
```

Exécuter ces programmes et expliquer la différence.

▷ **Exercice 11.** Variable d'accumulation

Dans l'éditeur, taper puis exécuter les lignes suivantes :

```
somme=0
for i in range(1,5):
    somme=somme+i
    print(somme)
```

Après exécution, que valent `i` et `somme` ?

Vérifier dans l'interpréteur (taper `i`).

▷ **Exercice 12.**

Dans l'éditeur, taper puis exécuter les lignes suivantes :

```
somme=0
for i in range(10,18):
    if i%3==0:
        somme=somme+i
```

Quelle est la valeur de `somme` après exécution ?

Vérifier.

▷ **Exercice 13.** Instruction conditionnelle `if ... else ...`

```
for n in range(20):
    if n%2==0:
        print(n,"est pair.")
    else:
        print(n,"est impair.")
```

Que fait le programme ci-dessus ?

Modifier ce programme pour afficher la liste des multiples de 7 compris entre 1 et 1000.

## B. Action

▷ **Exercice 14.** Factorielle

Calculer $10! = 1 \times 2 \times 3 \times \cdots \times 10$, puis 100! puis 1000!.

▷ **Exercice 15.** Somme des carrés

Calculer : $\sum_{k=10}^{100} k^2 = 10^2 + 11^2 + 12^2 + \cdots + 100^2$

▷ **Exercice 16.**

La liste des nombres inférieurs à 10 qui sont multiples de 3 ou de 5 est 3, 5, 6, 9. Leur somme est égale à 23.

Quelle est la somme des multiples de 3 ou de 5 inférieurs à 1 000 ?

▷ **Exercice 17.** Tables de multiplication

Écrire un programme qui demande à l'utilisateur d'entrer un entier $n$ grâce à l'instruction

```
n=int(input("Valeur de n : "))
```

et qui affiche la table de multiplication de $n$ jusqu'à 10 :

```
Valeur de n : 7
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
...
```

On pourrait supprimer les espaces en ajoutant l'option `sep=''` à l'instruction `print` :

```
print('Anti','constitutionnellement',sep='')
```

▷ **Exercice 18.** Nombres premiers

Tous les programmes suivants doivent être testés.

a. Écrire un programme qui demande un entier $n$ à l'utilisateur, puis qui affiche ses diviseurs stricts (*i.e.,* différents de 1 et de lui-même).
   Par exemple si $n = 20$ le programme doit afficher 2, 4, 5 et 10.
b. Modifier le programme précédent de façon à utiliser une boucle `while` et non une boucle `for`.
c. Modifier le programme précédent de façon à ce qu'il renvoie le plus petit diviseur de $n$ autre que 1. (Changer la condition).
d. Modifier le programme précédent de façon à ce qu'il renvoie `True` si $n$ est premier (*i.e.,* s'il n'a pas d'autres diviseurs que 1 et lui-même) et `False` sinon.
e. Afficher les nombres premiers compris entre 1 et 100 (on doit en obtenir 25).
f. Les entiers 2 005 001, 2 005 007 et 2 000 009 000 009 sont-ils premiers ? Si non, chercher leurs décompositions en facteurs premiers.

# Travaux Pratiques n°2
# Python: instructions de base

## I. Présentation de Pyzo

Nous utiliserons dorénavant `Pyzo`, un environnement de développement libre et gratuit pour `Python`.

▷ Créer, s'il n'existe pas déjà, un raccourci vers `Pyzo` sur le bureau. Cliquer dessus pour le lancer. S'assurer que la langue par défaut est le français, puis suivre le tutoriel.

Cet environnement de programmation présente plusieurs fenêtres. Leurs positions et leurs tailles sont modifiables à l'aide de la souris.

▷ Explorer les différentes possibilités offertes par le menu. On remarquera que les raccourcis habituels sont toujours disponibles (CTRL-O pour ouvrir un fichier existant, CTRL-N pour créer un nouveau fichier).

▷ Grâce au menu *paramètres*, désactiver les bulles d'aide et la complétion automatique.

### Les fenêtres outils

Les fenêtres en bas à droite contiennent différents outils. Il est possible d'en ajouter grâce au menu *outils*, ou de les supprimer en cliquant sur l'icône correspondant.

▷ Ne garder que l'espace de travail (*Workspace*) et le gestionnaire de fichier (*File Browser*). Positionner ce dernier sur votre répertoire `Python`, qui doit se trouver sur le disque `P`.

▷ Utiliser le gestionnaire de fichier de Pyzo pour vous positionner sur votre répertoire `Python`, y créer le répertoire `TP02` (clic droit sur `Python`, nouveau dossier), et se positionner dans celui-ci.

Ainsi les programmes créés seront sauvegardés dans ce répertoire.

### Le shell

La fenêtre du haut (ou en haut à droite) contient le *shell*, qui est l'interpréteur. Il est possible d'ouvrir plusieurs shells, accessibles par différents onglets.

Le raccourci CTRL-L efface le shell.

▷ Calculer $7 \times 11 \times 13$ dans le shell. Utiliser la flèche montante pour calculer ensuite $7 \times 11 \times 13 \times 17$ : il suffit de rappeler l'entrée précédente et d'ajouter la fin de la ligne.

On peut dans le shell demander la documentation sur une instruction ou une fonction :

```
>>> ? print
```

ou

```
>>> help(print)
```

**L'éditeur**

La fenêtre en bas à gauche (ou à gauche) contient l'éditeur. Là encore il est possible de travailler sur plusieurs fichiers à la fois, accessibles par des onglets.

*Il est fortement recommandé de sauvegarder* (CTRL-S) *l'éditeur avant même le début de l'écriture du programme.*

Choisir pour son programme une localisation (normalement il vous propose le répertoire actif `Python\TP02`) et un nom (`Ex01` par exemple, l'extension `py` est ajoutée automatiquement).

**Raccourcis utiles**

| | |
|---|---|
| CTRL-S | Sauvegarder le contenu de l'éditeur |
| CTRL-O | Ouvrir un fichier `py` existant |
| CTRL-N | Créer une nouvelle fenêtre d'édition |
| CTRL-E ou F5 | Exécuter le contenu de l'éditeur |
| CTRL-L | Effacer le shell |
| CTRL-I | Interrompre l'exécution (en cas de boucle infinie par exemple) |
| CTRL-R | Commenter la zone sélectionnée |
| CTRL-T | Décommenter la zone sélectionnée |
| et toujours : | |
| CTRL-C | Copier la zone sélectionnée |
| CTRL-X | Couper la zone sélectionnée |
| CTRL-V | Coller la zone copiée ou coupée |
| CTRL-Z | Annuler la dernière opération |
| CTRL-A | Sélectionner toute la zone |

▷ **Exercice 1.** Taper le programme suivant dans l'*éditeur*, le sauvegarder.

```
1 # Calcul de la factorielle
2
3 n=int(input("Entrez un entier naturel : "))
4
5 f=1
6 for k in range(n):
7     f=f*(k+1)
8
9 print(n,"!=",f,sep="")
```

Exécuter ce programme (F5 ou CTRL-E). Une fois que le nom est choisi, la sauvegarde est automatique à chaque exécution.

**Consigne valable pour toute la suite de ce TP et tous les TP suivants :**

**Sauvegarder tous les programmes écrits sous un nom explicite.**

Bien comprendre ce programme. Expliquer :

`#` (ligne 1)

**int** (ligne 3)

`k+1` (ligne 7)

`sep=""` (ligne 9)

Certaines réponses sont données dans la suite de ce TP.

## II. Entrées et sorties

Pour les entrées on utilise la fonction `input` qui renvoie une chaîne de caractères saisie par l'utilisateur et la stocke immédiatement :

```
A=input("Votre réponse : ")
```

Pour les sorties on utilise l'instruction `print(...)`, laquelle propose deux options : `sep` pour le séparateur (par défaut le caractère *espace*) et `end` pour la chaîne finale (par défaut le retour à la ligne `"\n"`) :

```
print(... , ...,        , sep='...', end='...')}
```

## III. Instructions conditionnelles

Les exercices suivants nécessitent l'utilisation des instructions **if**, **elif** et **else**.

▷ **Exercice 2.** À l'aide des instructions **input** et **print**, écrire un programme qui demande son nom à l'utilisateur et lui dit bonjour, en l'appelant par son nom.

Poursuivre son programme en demandant à l'utilisateur son année de naissance, et si son anniversaire a déjà eu lieu cette année.

Afficher ensuite la phrase "Tu as donc ... ans."

**Rappel**

**Ne pas oublier de commenter ses programmes grâce au symbole `#` !**

Ceci permet d'en dégager les grandes lignes, de séparer les étapes, d'expliquer le rôle d'une variable ou d'une instruction, etc... Ainsi vous le comprendrez si vous le reprenez plus tard, et de même pour un autre programmeur qui voudrait le prolonger.

Avec `Pyzo` il est possible de sélectionner toute une zone à la souris et de la commenter en pressant CTRL-R ou de la décommenter en pressant CTRL-T.

## IV. Instructions itératives

Les exercices suivants nécessitent l'utilisation des instructions de boucles **for** et **while**.

▷ **Exercice 3.** Reprendre le programme de l'exercice 1, le sauvegarder sous le nom `Ex03.py`.

Le modifier pour qu'il affiche la liste de toutes les factorielles de 0 à $n$!

Tester avec $n = 20$, $n = 100$ puis $n = 1000$.

▷ **Exercice 4.** Pour tout $n \in \mathbb{N}^*$ on pose : $S_n = \frac{4}{n}\sum_{k=0}^{n-1}\sqrt{1-\left(\frac{k}{n}\right)^2}$

a. Calculer à la main : $S_1 =$ $S_2 =$

b. Écrire un programme qui demande la valeur de $n$ à l'utilisateur, puis calcule et affiche la valeur de $S_n$.

Vérifier ce programme pour $n = 1$ et $n = 2$.

Tester ensuite ce programme pour de grandes valeurs de $n$. Avez-vous une conjecture sur la limite de $S_n$ ?

Oui. Je pense que

▷ **Exercice 5.** L'ordinateur choisit aléatoirement un nombre compris entre 1 et 100. Vous devez le deviner en lui proposant des nombres. Il répond à chaque essai «trop grand» ou «trop petit».

a. Écrire un programme permettant de jouer à ce jeu. Le tester.

- Pour obtenir un entier aléatoire compris entre 1 et 100 on importe au début du programme la fonction `randint` du module `random` :

```
from random import randint
...
a=randint(1,100)
```

- Ou pourra commencer par écrire juste un tour : le programme demande un nombre à l'utilisateur, il le compare à son nombre secret et affiche «trop grand», «trop petit» ou «bravo vous avez trouvé».

b. Ajouter à la fin l'affichage le nombre d'essais qui vous ont été nécessaires pour trouver le nombre.

▷ **Exercice 6.** La *Suite de Syracuse* est définie de la façon suivante :

- Le premier terme est un entier strictement positif $u_0$.
- Les termes suivants sont définis par la relation de récurrence :

$$\forall n \in \mathbb{N} \qquad u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

a. Vérifier à la main que pour $u_0 = 1, 2, 3, \ldots 10$, la suite finit toujours par arriver à 1.

1
2 →
3 → → → → → → →
4 →
5 →
6 →
7 →
8 →
9 →
10 →

b. Vérifier que si l'un des termes de la suite est égal à 1 alors la suite ne prend plus que 3 valeurs.

1 → → →

c. Écrire un programme demandant la valeur du premier terme, et affichant tous les termes suivants jusqu'à 1. Quel type de boucle utilisera-t-on ?

**Attention à la division**

On rappelle que `u/2` et `u//2` n'ont pas le même résultat, même si $u$ est un entier pair.

d. Raffiner le programme précédent pour afficher à la fin du calcul le nombre de termes qu'il aura fallu calculer pour arriver à 1. Ce nombre de termes est appelé *longueur* de la suite.

e. Modifier le programme pour obtenir l'affichage suivant :

```
Si u0 = 1 alors la longueur de la suite est 1
Si u0 = 2 alors la longueur de la suite est 2
Si u0 = 3 alors la longueur de la suite est 8
...
Si u0 = n alors la longueur de la suite est ...
```

f. La conjecture de Syracuse prédit que pour tout nombre entier $u_0$ la suite finit par arriver à 1.
Vérifier expérimentalement cette conjecture jusqu'à $10^5$.
On veillera à supprimer tout affichage superflu.

▷ **Exercice 7.** Soit $(a_n)$ la suite définie par $a_0 = 1$ et pour tout $n \in \mathbb{N}$, $a_{n+1} = 1 + \frac{1}{a_n}$.

a. Écrire un programme qui demande à l'utilisateur un entier $n$ puis qui calcule et affiche la valeur de $a_n$.
On remarquera qu'il suffit d'une unique variable $a$ (de type flottant) pour stocker les termes successifs de la suite.

b. Modifier votre programme pour qu'il affiche tous les termes intermédiaires, et utiliser ce programme pour établir une conjecture sur la limite de cette suite.

Je pense que

c. On définit maintenant la *suite de Fibonacci* par $u_0 = u_1 = 1$, et pour tout $n \in \mathbb{N}$ : $u_{n+2} = u_{n+1} + u_n$.
Écrire un programme qui affiche les $n$ premiers termes de cette suite.
On remarquera qu'il faut maintenant utiliser deux variables pour stocker les termes de la suite : une pour $u_n$ et une pour $u_{n-1}$.

d. Déterminer expérimentalement la limite de $\frac{u_{n+1}}{u_n}$.

Je pense que

# Travaux Pratiques n°3
# Listes et fonctions

## I. Listes

Taper les instructions suivantes dans le *Shell* de `Pyzo` (sans les commentaires). Elles présentent les commandes et opérations principales sur les listes.

**Définir une liste**

```
>>> L=[0,0,7]
```

L=

```
>>> M=list(range(11))
```

M=

**Définir une liste par compréhension**

```
>>> N=[x**2 for x in range(1,6)]
```

N=

```
>>> P=[2**p for p in range(20) if p%2==1]
```

P=

**Concaténation**

```
>>> L+N
```

```
>>> L*3
```

**Longueur d'une liste**

```
>>> len(P)
```

**Appeler un élément ou une sous-liste**

```
>>> L[2]
```

```
>>> N[1:4]
```

```
>>> P[0:8:2]  # avec un pas de 2
```

```
>>> P[:5]
```

```
>>> P[5:]
```

```
>>> P[2::3]
```

**Ajouter, supprimer un élément**

```
>>> N=N+[36]
```
N=

```
>>> N.append(49)
```
N=

```
>>> N.pop(3)
```
N=

```
>>> N.insert(4,289)
```
N=

▷ **Exercice 1.** Créer les listes suivantes.

`A:` la liste de tous les entiers de 10 à 20.

`B:` la liste de tous les entiers impairs de -15 à 15.

`C:` la liste des multiples de 7 compris entre 0 et 200.

`D:` la liste décroissante des entiers de 14 à 1.

`E:` la liste contenant 25 fois le chiffre 3.

`F:` la liste contenant un 1, puis deux 2, trois 3, etc jusqu'à vingt 20.
Pour ceci on peut initialiser la liste en posant `F=[]`, puis utiliser des boucles pour lui ajouter des éléments.

**Deux vérifications :**

Longueur de `F` :

Élément d'indice 150 de `F` :

## II. Fonctions

Le code ci-dessous contient un exemple de définition de fonction. Le saisir, le sauvegarder sous le nom `Exemple.py` et l'exécuter.

```
1  def factorielle(n):
2      """Renvoie la factorielle de n."""
3      f=1
4      for k in range(1,n+1):
5          f=f*k
6      return f
7
8  print(factorielle(5))
9  print(factorielle(0))
10 print(factorielle(1))
```

La chaîne de caractères suivant la définition (ligne 2 ) est la *documentation*, elle est affichée si on utilise la commande d'aide :

```
>>> help(factorielle)
```

Les *tests* (lignes 8 à 10) sont commentés (CTRL-R) dès que la fonction est au point. Ils couvrent un éventail de possibilités d'applications de la fonction, de façon à vérifier qu'elle fonctionne dans chacun des cas.

Si on veut plus tard remanier la fonction, on décommente les tests (CTRL-T) pour vérifier qu'elle fonctionne toujours.

Un autre façon de définir une fonction est l'utilisation d'une *fonction lambda*. Par exemple au lieu de

```
def f(x):
    """Renvoie le carré de x"""
    return x**2
```

on peut saisir :

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

Les fonctions sont tapées dans *l'éditeur*, éventuellement plusieurs par programme. La touche F5 (ou CTRL-E) exécute le programme dans le Shell.

On peut constater dans l'espace de travail (*Workspace*, en bas à droite de la fenêtre) que la fonction est définie et mémorisée : elle fait partie des variables déclarées. On devrait aussi y trouver les listes `A`, `B`... définies dans l'exercice 1.

**Consigne importante pour toute l'année**

La plupart des exercices proposés demande l'écriture d'un ou plusieurs programmes. Chaque programme doit être *sauvegardé* (CTRL-S) dans un répertoire convenable comme `Python\TP03`, sous un nom approprié comme `Ex05.py`.

De plus il faut *commenter* son programme (à l'aide du caractère `#`).

Ceci pour en dégager les grandes lignes, séparer les étapes, expliquer le rôle d'une variable, etc... Ainsi vous le comprendrez si vous le reprenez plus tard, et de même pour un autre programmeur qui voudrait le prolonger.

▷ **Exercice 2.**

Recopier le programme suivant.

```
def Occurrences(a,L):
    """..."""
    for k in range(len(L)):
        if a=k
            n=n+1
        return n
```

Ajouter une ligne de documentation sous la définition de la fonction, et au moins trois lignes de tests à la suite.

Corriger les cinq erreurs qui l'empêchent de fonctionner.

## III. Coefficients du binôme

▷ **Exercice 3.**

Cet exercice utilise la fonction `factorielle` déjà définie ci-dessus. Pour ceci récupérer le fichier `Exemple.py` et le sauvegarder sous le nom `Ex03.py`. Ne pas oublier de commenter les lignes de test.

a. Écrire une fonction qui reçoit deux entiers $n$ et $k$ et qui renvoie le coefficient du binôme $\binom{n}{k}$.

Tester la fonction en calculant $\binom{0}{0}$, $\binom{3}{1}$, $\binom{6}{9}$, $\binom{4}{k}$ pour $k$ allant de 0 à 4 (utiliser une boucle `for`), $\binom{50}{13}$ et $\binom{10\,000}{9\,997}$.

Pourquoi est-il préférable d'utiliser la division entière `//` plutôt que la division classique `/` ?

On propose maintenant une méthode rapide pour construire le triangle de Pascal.

b. Écrire une fonction `LigneSuivante` qui reçoit une liste de nombres $L = [a_0, a_1, \dots, a_n]$ et qui renvoie une liste $M = [b_0, b_1, \dots, b_{n+1}]$ définie par :

$$\forall k = 0, \dots, n+1 \qquad b_k = \begin{cases} a_0 & \text{si } k = 0 \\ a_{k-1} + a_k & \text{si } 1 \leqslant k \leqslant n \\ a_n & \text{si } k = n+1 \end{cases}$$

Par exemple `LigneSuivante([3,5,-1,0,7])` doit renvoyer :

c. Écrire maintenant un programme (et non une fonction) affichant une portion du triangle de Pascal.

Pour ceci commencer par demander à l'utilisateur un entier naturel $N$.

Définir `L=[1]`, et afficher `L`.

Itérer $N$ fois l'opération : remplacer `L` par `LigneSuivante(L)` et afficher `L`.

Constater le résultat, en testant des valeurs comme $N = 5$, $N = 10$, etc.

## IV. Nombres premiers

Un entier strictement positif est *premier* s'il admet exactement deux diviseurs, *i.e.,* s'il est strictement supérieur à 1 et n'admet pas d'autre diviseur que 1 et lui-même.

On teste si un nombre est divisible par un autre en utilisant l'opérateur `%`.

```
>>> if n%k==0:      # si k divise n
```

La propriété suivante (ou plutôt son corollaire) permet de gagner beaucoup de temps :

**Proposition.** *Soit $n \in \mathbb{N}^*$. Si $n$ n'est pas premier alors il admet un diviseur $k$ tel que $2 \leqslant k \leqslant \sqrt{n}$.*

Démonstration. Si $n$ n'est pas premier alors il existe deux entiers $a$ et $b$ supérieurs ou égaux à 2 tels que $n = ab$. Si $a$ et $b$ vérifient $a > \sqrt{n}$ et $b > \sqrt{n}$ alors par produit $ab > n$, ce qui est faux. Donc $a$ ou $b$ est inférieur ou égal à $\sqrt{n}$. □

**Corollaire.** *Si $n$ n'est divisible par aucun entier de 2 à $\lfloor \sqrt{n} \rfloor$ alors $n$ est premier.*

On *importe* la fonction `sqrt` (racine carrée) du module `math` en plaçant au début du programme l'instruction :

```
from math import sqrt
```

On rappelle que la partie entière $\lfloor x \rfloor$ est obtenue par l'instruction **int(x)**.

▷ **Exercice 4 : L'algorithme naïf.**

a. Écrire une fonction `est_premier` déterminant si un nombre entier est premier. Cette fonction recevra un entier et renverra **True** ou **False**.

b. Tester cette fonction sur les entiers de 1 à 20 :

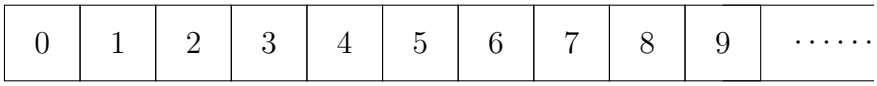
```
def est_premier(n):
    ....

for i in range(1,21):
    if est_premier(i):
        print(i,end=' ')
print() # pour le retour à la ligne
```

c. Combien de nombres premiers sont inférieurs à 1000 ?

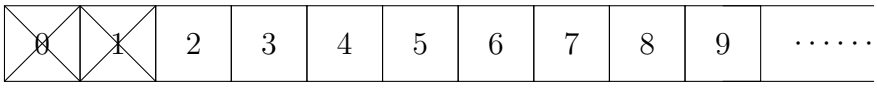
d. Déterminer le plus petit nombre premier supérieur à $10^{10}$.

▷ **Exercice 5 : Le crible d'Ératosthène.**

La méthode d'Ératosthène permet de déterminer tous les nombres premiers inférieurs à $N$, où $N$ est un entier fixé. Pour ceci on écrit tous les nombres entiers de 2 à $N$ :

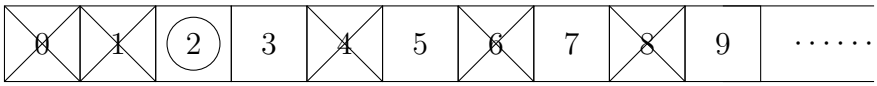



On barre 0 et 1 qui ne sont pas premiers.

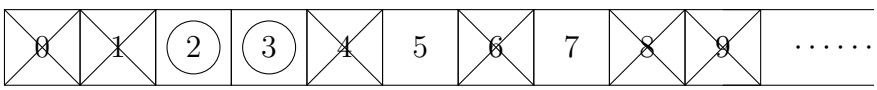



On itère les opérations suivantes :

- Le premier nombre non barré est 2, il est donc premier. Tous ses multiples stricts (4, 6, 8...) sont composés, donc on les barre :

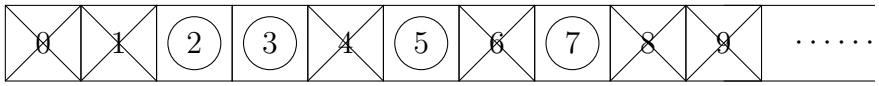



- Ensuite, comme le 3 n'est pas barré, il est premier et on barre tous ses multiples stricts :



- On saute le 4 qui est barré, on arrive au 5, on barre tous ses multiples stricts.
- Etc jusqu'à...

À l'issue de l'algorithme les nombres non barrés sont les nombres premiers.




On représente le tableau par une liste de booléens : `C=[True,True,...]`. Au début il ne contient que la valeur **True**. À la fin la valeur de chaque composante `C[n]` est **True** si $n$ est premier, **False** si $n$ est composé.

a. Programmer l'algorithme du crible d'Ératosthène. Vérifier que la fonction est correcte pour les nombres premiers inférieurs à 20.

```
def Crible(N):
    ....

C1=Crible(20)
for n in range(20):
    if C1[n]:           # Test : n est-il premier ?
        print(n,end=' ')
print()
```

b. Retrouver le nombre de nombres premiers inférieurs à 1000.

▷ **Exercice 6 : Comparaison des algorithmes.**

On souhaite comparer le temps d'exécution des deux algorithmes.

On mesure le temps grâce à la fonction `clock` du module `time`. Le résultat est donné en secondes, il s'agit du temps de travail du processeur.

```
from time import clock
t1=clock()
... # Exécution de l'algorithme
t2=clock()
print("Temps :",t2-t1)
```

On remarque que le tableau `C` déterminé par le crible d'Ératosthène peut être obtenu facilement avec le premier algorithme de la façon suivante :

```
C=[]
for n in range(N+1):
    C.append(est_premier(n))
```

a. Dans une nouvelle fenêtre, importer les module `time`, puis définir $N = 100$. Mesurer le temps de calcul du tableau `C` en utilisant les deux méthodes, puis afficher ces temps.

Temps pour $N = 100$ :

b. Recommencer avec $N = 10^4$, $N = 10^5$.

Temps pour $N = 10^4$ :

Temps pour $N = 10^5$ :

L'algorithme le plus rapide est

## V. Compléments

Les trois exercices de cette section peuvent être traités dans l'ordre de votre choix.

▷ **Exercice 7 : Nombres parfaits et amis.**

a. Écrire une fonction recevant un entier $n$ positif, et renvoyant la liste de ses diviseurs autres que lui-même.

b. Un nombre est parfait s'il est égal à la somme de tous ses diviseurs autres que lui-même. Trouver tous les nombres parfaits inférieurs à $10^4$.

c. Deux nombres distincts sont amis si chacun est égal à la somme des diviseurs stricts de l'autre. Trouver tous les couples de nombres amis inférieurs à $10^4$.

▷ **Exercice 8 : Nombres premiers jumeaux.**

Des nombres premiers *jumeaux* sont des entiers premiers distants de 2 unités, comme par exemple $(3, 5)$, $(5, 7)$ et $(11, 13)$.

Une conjecture affirme qu'il existe une infinité de nombres premiers jumeaux. Elle n'a pas été démontrée à ce jour.

Afficher tous les couples de nombres jumeaux inférieurs à $2^{20}$, et déterminer le nombre de tels couples.

▷ **Exercice 9. Conjecture de Goldbach**

Cette conjecture affirme que tout nombre pair supérieur à 3 est somme de deux nombres premiers. Elle n'a pas été démontrée à l'heure actuelle.

Afficher tous les nombres pairs inférieurs à 100 comme somme de deux entiers premiers.

```
4 = 2 + 2
6 = 3 + 3
8 = 3 + 5
...
```

Vérifier que la conjecture de Goldbach est vraie pour tout entier pair inférieur à $2^{20}$.

# VI. Indications

▷ **Exercice 3.**

b. La structure de la fonction `LigneSuivante` :

```
def LigneSuivante(L):
    M=[L[0]]
    for k in range( ??? ):
        M.append( ??? )
    M.append(L[-1])
    return M
```

▷ **Exercice 4.**

L'algorithme naïf en pseudo-code :

**Entrées** : $n$ (entier)
**Si** $n < 2$ **alors**
  **Résultat** : *à compléter*
**FinSi**
**Pour** $k$ allant de 2 à $\lfloor\sqrt{n}\rfloor$ **faire**
  **Si** $k$ divise $n$ **alors**
    **Résultat** : *à compléter*
  **FinSi**
**FinPour**
**Résultat** : *à compléter*

▷ **Exercice 5.**

L'algorithme du crible d'Ératosthène en pseudo-code :

**Entrées** : $N$ (entier)
$C$ **reçoit** [False,False]+(N-2)*[True]
**Pour** $n$ allant de 2 à $\left\lfloor\sqrt{N}\right\rfloor$ **faire**
  **Si** $C[n]$ **alors**
    ($n$ est premier : on barre ses multiples)
    **Pour** $k$ allant de ? à ? [avec un pas de ?] **faire**
      C[ ?] **reçoit** False
    **FinPour**
  **FinSi**
**FinPour**
**Résultat** : C

# Travaux Pratiques n°4
# Graphiques

**But de cette séance :** Tracer des graphiques en deux dimensions.

## I. Matplotlib

Le sous-module pyplot du module matplotlib permet de créer des graphiques. Il est importé au début du programme, ce qui charge en mémoire les instructions `plot`, `show`, etc.

```
from matplotlib.pyplot import *

plot([5,0],[3,2])             # un segment
plot([1,7,5,1],[2,2,4,1])     # une ligne polygonale
show()
```

▷ Tester ce programme.

La commande `show` ouvre une fenêtre contenant le graphique. Aller la chercher dans la barre Windows en bas de l'écran. Il faudra la fermer pour lancer le programme suivant.

**À retenir**

Pour tracer une ligne polygonale reliant les points de coordonnées $(x_1, y_1)$, $(x_2, y_2)$, $\ldots$, $(x_n, y_n)$ :

```
x=[x1,x2, ..., xn]   # les abscisses
y=[y1,y2, ..., yn]   # les ordonnées
plot(x,y)
show()
```

Quelques options de la commande `plot` :

```
plot(x,y,'b:',linewidth=4,label='Ma courbe')
   # couleur bleue ('b'), pointillés (':'), épaisseur 4, légende.
```

Autres couleurs : `'r'`, `'g'`, `'k'`, etc (pour rouge, vert, noir, etc). Par défaut les couleurs sont automatiquement modifiées pour chaque courbe.

Autres marqueurs : `'.'`, `'o-'`, `'--'` (points, gros points reliés, tirets).

On écrit `'b:'` pour condenser `'b',':'`.

Pour obtenir la liste complète des options :

```
>>> help(plot)
```

Il est également possible de configurer les axes, d'ajouter un titre, d'afficher la légende en choisissant son positionnement, etc. Ceci grâce aux commandes suivantes, à placer entre les instructions `plot` et `show`.

```
axis("equal")                # repère orthonormé
axis([a,b,c,d])              # bornes de la fenêtre : a<x<b et c<y<d

grid()                       # affichage d'une grille

axhline(color="black")       # affichage des axes
axvline(color="black")
xlabel("x")                  # noms des axes
ylabel("y")

legend(loc="upper left")     # Affichage et position de la légende

title("Un joli dessin")      # titre de la figure
```

▷ **Exercice 1.**

On rappelle que les racines $n$-èmes de l'unité ont pour coordonnées $\left(\cos k\frac{2\pi}{n}, \sin k\frac{2\pi}{n}\right)$ pour $k$ allant de 0 à $n-1$. On importe les fonctions et constantes mathématiques (dont `cos`, `sin` et `pi`) grâce à :

```
from math import *
```

a. Tracer le polygone régulier à 10 côtés formé par les racines 10-èmes de l'unité.

b. Créer un graphique contenant les polygones réguliers formés par les racines $2^p$-èmes de l'unité, pour $p$ allant de 2 à 10.

   Ajouter les noms des polygones (`label="p="+str(p)`) et afficher la légende en haut à droite (ce qui peut nécessiter d'agrandir la fenêtre d'affichage).

   À partir de combien de points peut-on considérer qu'on a une bonne représentation du cercle trigonométrique ? (Il est possible d'agrandir la fenêtre)

   Réponse :

c. Définir $n = 24$ et $m = 7$.

   Tracer ensuite le polygone reliant les points de coordonnées $\left(\cos k\frac{2m\pi}{n}, \sin k\frac{2m\pi}{n}\right)$ pour $k$ allant de 0 à $n$.

   Ajouter le titre «un joli dessin».

   On pourra modifier les valeurs de $m$ et $n$.

## II. Numpy

Le module `numpy` permet d'utiliser des matrices. Il définit un nouveau type : `array`, pour représenter des tableaux multidimensionnels comme des vecteurs ou des matrices.

Pour l'instant nous utiliserons juste la fonction `linspace` et les fonctions vectorielles.

La fonction `linspace(a,b,p)` renvoie un tableau contenant $p$ réels uniformément répartis de $a$ à $b$.

▷ Entrer les instructions suivantes dans le *Shell* et les comprendre :

```
>>> from numpy import *
>>> linspace(0,10,21)
>>> linspace(5,11,3)
>>> C=linspace(0,10)          # par défaut la troisième variable vaut 50
>>> len(C)                    # toujours valable
>>> C[25]                     # toujours valable
```

En général, une fonction comme la fonction `sin` du module `math` associe un réel à un réel. Avec les fonctions vectorielles, il est également possible d'utiliser une liste de réels $L = [x_1, \ldots, x_n]$ comme variable d'entrée. La fonction renvoie alors la liste $f(L) = [f(x_1), \ldots, f(x_n)]$.

Renouveler le Shell et saisir les instructions suivantes.

```
>>> from math import sin, pi
>>> import numpy as np        # importation de numpy sous le nom np
>>> sin(pi/6)                 # fonction sinus du module math
>>> np.sin(pi/6)              # fonction sinus du module numpy
                              # une différence ?
>>> L=np.linspace(0,10,11)
>>> sin(L)
>>> np.sin(L)                 # et là ?

>>> L*pi/2                    # on constate qu'on peut multiplier
                              # un tableau numpy par un scalaire
>>> np.sin(L*pi/2)            # Étranges valeurs...
```

**Remarque sur les flottants.** La valeur d'un flottant est toujours approximative. Pour cette raison, il ne faut pas demander de test d'égalité entre flottants, et *a fortiori* pas de test de nullité ($a = b$ équivaut à $a - b = 0$).

Quel est cet ordre de grandeur d'un réel presque nul en Python ? Comment tester la nullité d'un flottant ?

| | | |
|---|---|---|
| `if x==0:` | pourra s'écrire | `if` |

**Remarque sur l'importation des modules.** Deux méthodes sont possibles. La première importe le module lui-même, ce qui définit un nouvel espace de nom séparé de l'espace principal, il faut alors préfixer les fonctions du module par son nom :

```
import math
math.sin(math.pi/6)
```

Ce nom peut d'ailleurs être modifié :

```
import math as m
m.sin(m.pi/6)
```

La seconde méthode importe toutes les fonctions du module :

```
from math import *
sin(pi/6)
```

Il faut alors veiller aux conflits possibles si deux modules contiennent des fonctions de même nom. Par exemple la fonction `sin` est présente à la fois dans le module `math` et dans le module `numpy`, tout comme les autres fonctions mathématiques. Celle du module `numpy` est préférable car elle est vectorielle. Il est donc inutile d'importer le module `math` si on importe le module `numpy`.

Attention toutefois, cette seconde méthode peut surcharger inutilement la mémoire et ralentir les processus.

**Remarque sur les fonctions vectorielles.** Les instructions ci-dessous montrent bien la différence entre les listes de Python (**list**) et les tableaux numpy (`ndarray`). Les tester dans le Shell (sans les commentaires).

```
>>> from numpy import *
>>>
>>> LA=[0,2,4]
>>> LB=[1,1,5]
>>> type(LA)     # quel est le type de LA ?
>>>
>>> TA=array([0,2,4])   # ou TA=array(LA)
>>> TB=array([1,1,5])   # ou TB=array(LB)
>>> type(TA)     # quel est le type de LA ?
>>>
>>> LA+LB
>>> 5*LA
>>>
>>> TA+TB
>>> 5*TA
```

On peut vérifier que les fonctions arithmétiques (`+ - * / ** // %`) sont vectorielles, ce qui permet de définir des fonctions avec ces opérateurs et de les appliquer à une liste.

Vérifier ceci en tapant les lignes suivantes dans le Shell :

```
>>> L=linspace(0,6,7)
>>> type(L)
>>> f=lambda x:x**2-1          # une fonction du second degré
>>> L
>>> f(L)
```

On obtient `f(L)=`

Ainsi, pour tracer la courbe représentative de $f$ on peut utiliser

```
>>> plot(L,f(L))
```

en ajoutant des points. Pour ceci il suffit de remplacer le 7 dans la définition de L par 200 ou même un plus grand nombre.

**À retenir**

Pour tracer la courbe d'une fonction $f$ sur un intervalle $[a, b]$ :

```
x=linspace(a,b,200)
y=f(x)
plot(x,y)
show()
```

Télécharger le programme `ModeleGraphique.py` depuis le site `prepabellevue.org`, rubrique `PCSI2/Informatique/Travaux Pratiques`.

Le sauvegarder dans votre répertoire `Python`.

Il servira de base pour la plupart des graphiques que nous ferons dans ce TP et les suivants.

Il suffira de le charger sous `Pyzo`, de le copier-coller dans un autre fichier, ou de le sauvegarder sous un autre nom (`Exo2.py` par exemple). Puis on le modifiera pour avoir le graphique désiré.

▷ **Exercice 2.** Soit $f(x) = x\sqrt{1-x^2}$.

a. Tracer sur un même graphique la courbe représentative de la fonction $f$ et le cercle trigonométrique.
   Vérifier que les axes, la grille, la légende sont bien présents.
   Utiliser un repère orthonormé, et afficher la zone $[-1{,}2\,;\,1{,}2] \times [-1{,}2\,;\,1{,}2]$.

b. Supprimer les axes, la grille et toute autre fioriture, et afficher les deux courbes en noir, avec une épaisseur de 16 points.

# III. Applications

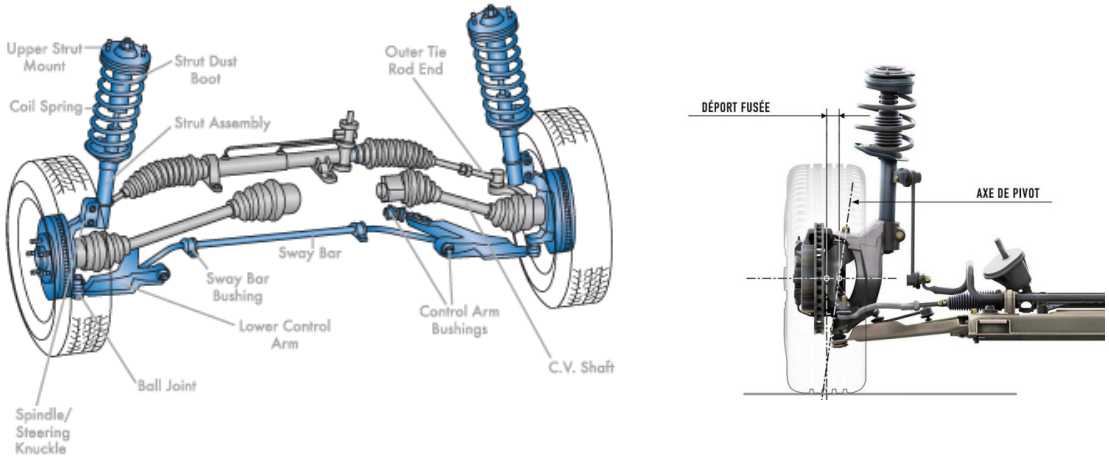
▷ **Exercice 3 : Modélisation d'une suspension de véhicule.**

**But :** Tracer le diagramme de Bode d'un système.

Afin de limiter l'impact des irrégularités de la surface d'une route sur le véhicule qui la parcourt, l'utilisation d'un système de suspensions est indispensable. Il permet de limiter les risques de rupture et/ou d'usure excessive, améliore le confort de conduite et maintient le contact entre les roues et le sol malgré ses irrégularités. Cette condition est indispensable à la tenue de route.

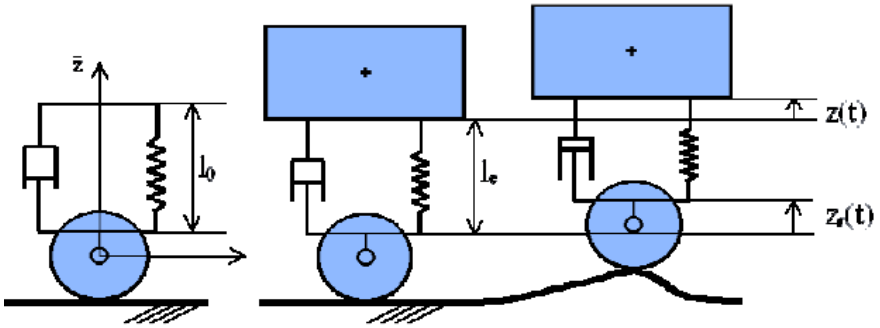
Ainsi, la suspension constitue un dispositif de liaison entre des « masses non suspendues » (roues, système de freinage...) et des « masses suspendues » (châssis, moteur...). Elle se compose d'un ressort et éventuellement d'un amortisseur (cf. figure 1).

FIGURE 1 – Suspension avant automobile




Le quart de véhicule est modélisé par une masse $M$, la suspension est modélisée par un système ressort-amortisseur, $K_{\text{susp}}$ représente la raideur du ressort et $f_{\text{susp}}$ le coefficient de frottement visqueux de l'amortisseur (trous calibrés de passage de l'huile au travers du piston de l'amortisseur hydraulique).

FIGURE 2 – Modèle quart de véhicule



Une étude mécanique réalisée sur le modèle du quart de véhicule nous donne l'équation différentielle liant le déplacement $z(t)$ du châssis du véhicule (grandeur de sortie) au déplacement $z_r(t)$ imposé sur l'axe de roue par le profil de la route (grandeur d'entrée).

$$K_{\text{susp}} z_r(t) + f_{\text{susp}} \frac{\mathrm{d}z_r(t)}{\mathrm{d}t} = M \frac{\mathrm{d}^2 z(t)}{\mathrm{d}t^2} + f_{\text{susp}} \frac{\mathrm{d}z(t)}{\mathrm{d}t} + K_{\text{susp}} z(t)$$

La fonction de transfert du système s'écrit donc :

$$H(p) = K \frac{1 + \tau p}{1 + \frac{2\xi}{\omega_0} p + \frac{1}{\omega_0^2} p^2}$$

On donne les valeurs numériques suivantes :

$$K = 1 \qquad \omega_0 = 70{,}71\text{s}^{-1} \qquad \tau = 0{,}004\text{s} \qquad \xi = 0{,}141$$

L'objectif est de tracer les diagrammes de Bode du système et de déterminer pour quelles fréquences la suspension risque d'être endommagée.

**Les complexes en** Python**.** (à tester dans le Shell)

```
>>> z=1j
>>> z**2
>>> z=1+1j      # variante : z=complex(1,1)
>>> z**2
>>> from numpy import *
>>> angle(z)    # argument, en radians
>>> absolute(z) # module
```

a. Importer les bibliothèques numpy et matplotlib.pyplot.
b. Définir les constantes puis la fonction de transfert sous la forme d'une fonction Python.
c. Définir une variable `puissance_w` qui est la liste des puissances de 10 à considérer en abscisse. On prendra de 0 à 4 avec un pas de 0,01.
d. Définir une variable `W` définie comme $10^{\texttt{puissance\_w}}$. Cette variable sera utilisée pour la définition de l'axe des abscisses.

Pour la suite on rappelle que la phase est l'argument de la fonction de transfert, et le gain est : $20 \log |H(j\omega)|$, où $|H(j\omega)|$ est le module de la fonction de transfert.

e. Définir les ordonnées de notre graphique, à savoir le gain et la phase.
f. Tracer les deux diagrammes de Bode (gain et phase) dans la même fenêtre. Utiliser pour ceci l'instruction `subplot`, voir page suivante.

   Attention, pour que l'axe des abscisses soit en échelle logarithmique, on n'utilise pas `plot` mais `semilogx`.

   Tracer les courbes en rouge. Ajouter des noms pour les axes et une légende.

```
# Pour tracer deux graphiques dans la même fenêtre

subplot(2,1,1)   # Deux lignes, une colonne, graphique 1
...
plot(X,Y1)
...

subplot(2,1,2)   # Deux lignes, une colonne, graphique 2
...
plot(X,Y2)
...

show()
```

g. Y a-t-il un risque d'endommagement de la suspension pour certaines sollicitations sinusoïdales ?

Identifier la zone de risque et ajouter du texte pour l'indiquer au niveau des graphes.

```
# Pour ajouter du texte sur le graphique

text(x,y,"Texte",horizontalalignment='center')
```

▷ **Exercice 4 : Vendée Globe.**

Soit $f(x) = \dfrac{\sin x}{\alpha + \cos x}$ où $\alpha \in ]1, +\infty]$.

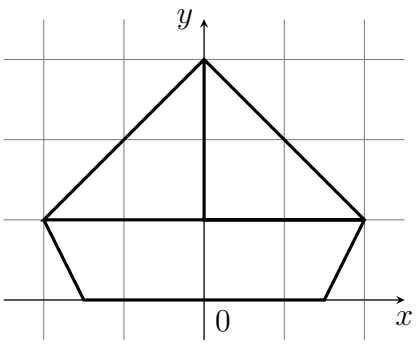
a. Poser $\alpha = 2$. Tracer la courbe de $f$ dans un repère orthonormé sur l'intervalle $[-15, 15]$. Imposer la couleur bleue.

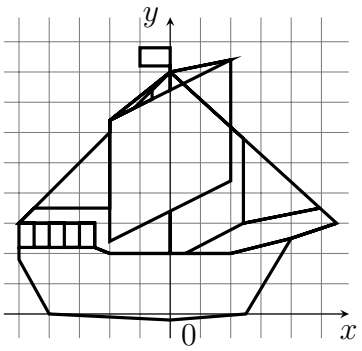
b. Supprimer tous les éléments autres que la courbe. Le décor est planté.

Tester d'autres valeurs de $\alpha$. On constatera que l'on rend la mer un peu plus calme en augmentant la valeur de $\alpha$, et un peu plus houleuse en l'abaissant (mais il faut toujours $\alpha > 1$).

On ajoute maintenant des bateaux au graphique. Ces bateaux doivent voguer sur les flots, donc être tangents à la courbe de $f$.

Un bateau sera représenté par une unique ligne polygonale. On commence par en construire un dont le point central en bas est l'origine du repère, comme ci-dessous. On compose soi-même son propre bateau !




c. Définir `xb` et `yb` la liste des abscisses et des ordonnées du bateau, avec une échelle quelconque. Afficher le bateau.

```
xb=[...]
yb=[...]
plot(xb,yb)
```

Pour placer le bateau sur la courbe de $f$, au niveau d'un point d'abscisse $x_0$ et d'ordonnée $y_0 = f(x_0)$, il faut appliquer à chaque point de coordonnées $(x, y)$ les transformations successives suivantes :

$$\begin{pmatrix} x \\ y \end{pmatrix} \xmapsto{\text{homothétie}} \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \xmapsto{\text{rotation}} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \xmapsto{\text{translation}} \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}$$

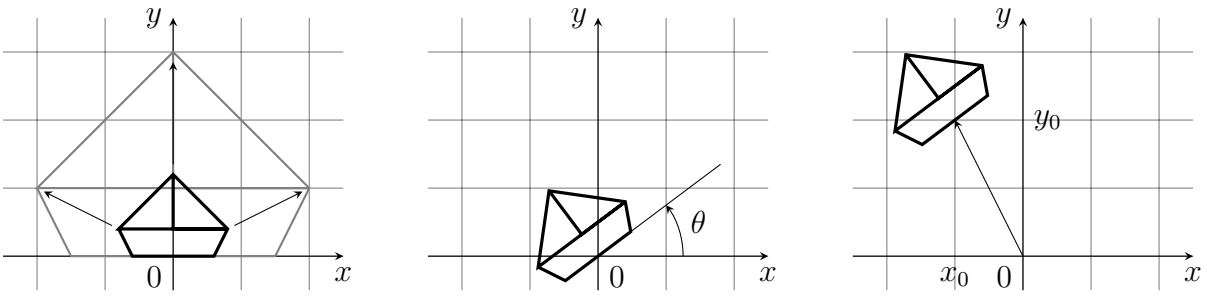
Plus précisément ces transformations sont :

- une homothétie $x_1 = hx$ et $y_1 = hy$, où $h$ est un coefficient défini à l'avance.
- une rotation d'angle $\theta = \arctan f'(x_0)$, qui s'obtient par les formules :

$$x_2 = x_1 \cos\theta - y_1 \sin\theta \qquad \text{et} \qquad y_2 = x_1 \sin\theta + y_1 \cos\theta$$

  La fonction `arctan` est fournie par `numpy`.
  La dérivée de $f$ est : $f'(x) = \dfrac{1 + \alpha \cos x}{(\alpha + \cos x)^2}$
- une translation de vecteur $(x_0, y_0)$ : $x_3 = x_2 + x_0$ et $y_3 = y_2 + y_0$.

Homothétie Rotation Translation




Il est possible d'appliquer ces trois transformations sur chacun des points du bateau. Mais on peut aussi les appliquer directement sur les listes `xb` et `yb`. Pour ceci il faut les avoir définies comme des tableaux `numpy` et non comme des simples listes :

```
xb=array([0,3,2,...])
```

d. Définir le coefficient $h$ de l'homothétie.
   Choisir une valeur arbitraire de $x_0$, puis afficher votre bateau sur le point d'abscisse $x_0$ de la courbe de $f$. C'est-à-dire appliquer les trois transformations aux listes `xb` et `yb` puis afficher le résultat.
   Ensuite, grâce à une boucle afficher plusieurs bateaux sur la courbe de $f$.
   On peut par exemple choisir $x_0$ allant de $-14$ à 14 avec un pas de 2.
   Le rendu sera meilleur si les bateaux sont tracés avant la courbe de $f$.
   Modifier la valeur de $\alpha$ pour rendre la mer plus ou moins agitée et vérifier que le bateau est toujours dessus.

e. Créer maintenant une vidéo où le bateau vogue sur l'eau :

```
for x0 in range(-15,16):
    clf()          # efface la figure
    ...
    plot(...)
    ...
    pause(1e-3)    # temps très court

show()
```

# Travaux Pratiques n°5
# Manipulation de fichiers

Le but de ce TP est d'apprendre à lire, créer ou modifier des fichiers.

## I. Le module os

▷ Pour commencer ouvrir le répertoire Python. Ne pas créer de répertoire `TP05` !

Laisser simplement la fenêtre ouverte sur le bureau Windows.

▷ Ouvrir Pyzo, importer le module `os` (operating system) :

```
>>> import os
>>> os.getcwd()
```

L'instruction `getcwd` indique le répertoire courant de travail (Current Working Directory).

On peut accéder à l'adresse complète du répertoire Python via le bureau Windows : cliquer sur la barre d'adresse et copier celle-ci. À la maison on devrait obtenir quelque chose comme `C:\User\Alphonse\Mes documents\Python`. Sur les ordinateurs du lycée, le répertoire Python est dans le disque `P`.

▷ Changer le répertoire de travail grâce à l'instruction `chdir` (Change directory).

```
>>> os.chdir("P:\\Python")
>>> os.getcwd()
```

▷ Créer le répertoire `TP05` à l'aide de l'instruction `mkdir` (make directory) et le choisir comme répertoire courant.

```
>>> os.mkdir("TP05")
>>> os.chdir("TP05")
>>> os.getcwd()    # vérification
```

▷ Constater dans l'explorateur de Windows que le répertoire est bien créé.

Tous les fichiers de ce TP seront créés et sauvegardés dans ce répertoire, donc on ne modifiera plus le répertoire courant de travail.

**Les fonctions principales du module `os` :**

| | |
|---|---|
| `getcwd()` | Renvoie le répertoire courant de travail |
| `chdir(chemin)` | change de répertoire courant |
| `mkdir(rep)` | crée le répertoire `rep` |
| `rmdir(rep)` | supprime le répertoire `rep` |
| `listdir(chemin)` | affiche la liste des fichiers contenus dans le répertoire `chemin` (par défaut le répertoire courant) |
| `remove(fichier)` | supprime `fichier` |
| `rename(source,dest)` | renomme `source` en `dest` |

## II. Fichiers texte

Les fichiers `txt` sont des fichiers ne contenant que du texte, sans mise en forme, mais avec les retours à la ligne et éventuellement des tabulations. Sous Windows ils sont ouverts par l'application Bloc-notes (Notepad en anglais).

Pour accéder à un fichier `txt` en Python on l'ouvre grâce à la fonction **open**. En fin d'utilisation on le ferme grâce à la méthode `close`.

Un fichier peut être ouvert en lecture (**open** avec l'option `'r'`) ou en écriture (**open** avec l'option `'w'`). Cette option permet également de créer un fichier. Il existe d'autres options comme `'a'` (append) pour l'ajout à la fin du fichier.

**Ouverture d'un fichier en écriture :**

▷ Taper les instructions ci-dessous (sans les commentaires) dans le Shell.

```
>>> montxt=open('Test.txt','w')
```

Comme le fichier n'existait pas il est créé.

```
>>> montxt.write("J'écris un texte,\n et il va être sauvegardé.\n")
    # Python renvoie le nombre de caractères écrits
>>> montxt.write("C'est super !")
>>> montxt.close()
```

Ce n'est qu'à l'exécution de la fonction `close` que le fichier est physiquement créé (ou modifié).

▷ Retourner dans la fenêtre de l'explorateur de Windows contenant le répertoire Python, double-cliquer sur `test.txt` pour vérifier son contenu. Le bloc-notes doit s'ouvrir.

Ne pas oublier la chaîne de caractère `\n` pour le retour à la ligne !

**Ouverture en lecture :**

▷ Taper les instructions ci-dessous, toujours dans le Shell :

```
>>> montxt=open('Test.txt','r')
>>> S=montxt.read()
>>> print(S)
>>> montxt.close()
```

Penser à fermer le fichier pour éviter les conflits.

Un autre moyen courant d'utiliser un fichier est de le traiter ligne par ligne, donc de parcourir les lignes.

▷ Taper l'exemple ci-dessous dans *l'éditeur* :

```
montxt=open('Test.txt')    # L'option 'r' est sélectionnée par défaut
k=0
for L in montxt:
    k=k+1
    print("Ligne",k,": ",L)
montxt.close()
```

**À retenir**

Écriture dans un fichier

```
montxt=open('fichier.txt','w')
...
montxt.write('texte...')
...
montxt.close()
```

Lecture d'un fichier

```
montxt=open('fichier.txt','r')
...
S=montxt.read() # ou
L=montxt.readline()
...
montxt.close()
```

**Méthodes principales d'utilisation d'un fichier :**

| | | |
|---|---|---|
| `read()` | lit le fichier en entier | renvoie une chaîne de caractères |
| `read(n)` | lit $n$ caractères | renvoie une chaîne de longueur au plus $n$, vide si tout le fichier a été lu. |
| `readline()` | lit une ligne | renvoie une chaîne de caractère, vide si le fichier est fini |
| `readlines()` | lit toutes les lignes | renvoie la liste de toutes les lignes sous forme de chaînes de caractères. |
| `write(S)` | écrit la chaîne `S` | |
| `writelines(L)` | écrit les lignes de la liste `L` | |
| `close()` | enregistre le fichier | |

Toutes ces méthodes s'appliquent à un objet de type fichier comme `montxt` : `L=montxt.readline()`, `montxt.write("Hello")`...

**Remarque sur les méthodes :**

Une *méthode* est une syntaxe particulière aux langages orientés objets. Par exemple :

```
>>> L=[6,8,3,5,6]
>>> sorted(L) # fonction de tri
[3,5,6,6,8]
```

```
>>> L=[6,8,3,5,6]
>>> L.sort()  # méthode de tri
              # L est triée
```

L'instruction `sorted` est une *fonction* qui a pour paramètre la variable $L$.

L'instruction `sort()` est une *méthode* appliquée à l'objet $L$.

**À retenir**

Utilisation d'une méthode :

```
objet.methode()
```

Utilisation d'une fonction :

```
fonction(objet)
```

Avec arguments :

```
objet.methode(arg)
```

Avec arguments :

```
fonction(objet,arg)
```

## III. Chaînes de caractères

**Quelques fonctions et méthodes s'appliquant à une chaîne de caractères `S` :**

| | | |
|---|---|---|
| `len(S)` | longueur de `S` | |
| `S[i]` | élément d'indice `i` de `S` | `S[-1]` pour le dernier élément |
| `S[i:j]` | sous-chaîne de `S[i]` à `S[j-1]` | `S[:j]`, `S[i:]` pour le début, la fin |
| `c in S` | teste si `c` est dans `S` | |
| `S.split()` | renvoie la liste des mots de `S` | |
| `S.upper()` | transforme les minuscules en majuscules | `S.lower()`, `S.capitalize()` existent aussi |

Pour la liste complète : **`help(str)`**. Les méthodes `count`, `index`, `replace`, `join` peuvent être utiles également.

▷ Tester les méthodes données ci-dessus, en tapant les lignes suivantes dans le Shell (sans les commentaires) :

```
>>> montxt=open("Test.txt")
>>> Texte=montxt.read()
>>> montxt.close()

>>> print(Texte.upper())
>>> "super" in Texte
>>> Texte.index("super")   # indice du premier caractère de "super"
>>> Texte.split()
```

La méthode `split` sépare une chaîne de caractères en une liste de chaînes de caractères selon un caractère particulier. Par défaut le délimiteur est l'espace (`S.split()`) mais il est possible d'en spécifier un autre.

▷ Tester (sans les commentaires) :

```
>>> Ligne="1;5;6.5;200"
>>> L1=Ligne.split(";")
>>> L1
>>> type(L1)      # Quel est le type de L1 ?
>>> type(L1[0])   # Quel est le type d'un élément de L1 ?
>>> Liste=[float(x) for x in L1]
>>> Liste
```

On peut aussi essayer `Ligne.split("5")`.

# IV. Exercices

▷ Télécharger depuis le site `prepabellevue.org` le fichier compressé `TP05-Fichiers.zip`, en utilisant la fonction « enregistrer la cible du lien sous ». Le décompresser et placer dans le répertoire `TP05` tous les fichiers qu'il contient.

▷ **Exercice 1.**

a. Écrire une fonction `Occurrences(a,S)` comptant le nombre d'occurrences d'un caractère `a` dans une chaîne `S`.

b. Ouvrir le ficher `LaFontaine.txt` à l'aide de Python. Stocker son contenu, par exemple dans la variable `Texte`.

Convertir Toutes ses majuscules en minuscules.

c. Compter le nombre d'occurrences de chaque lettre de l'alphabet dans ce texte.

Pour ceci il peut être utile de noter `Alphabet="azert..."` la chaîne de toutes les lettres de l'alphabet, non obligatoirement classées. Il suffira alors d'écrire :

```
for x in Alphabet:
    print(x,"apparaît",Occurrences(x,Texte),"fois.")
```

d. Créer un fichier `Compte.csv` contenant 26 lignes, toutes de la forme `"x,k"` où $k$ est le nombre d'occurrences de `x`.

Attention ! Ces lignes doivent être des chaînes de caractères (et il faut leur ajouter le retour à la ligne `'\n'`).

Rappel : la fonction convertissant un nombre (5) en chaîne de caractères ("5") est :

Réponse :

*On crée un fichier* `csv` *pour pouvoir le manipuler avec LibreOffice ou Excel. Les colonnes d'un tel fichier sont généralement séparées par des virgules.*

e. Sous Windows double-cliquer sur le ficher `Compte.csv` pour voir le nombre d'occurrences de chaque lettre. Trier les lignes pour voir les lettres les plus fréquentes et les plus rares.

*Pour trier des lignes selon les éléments d'une colonne sous LibreOffice ou Excel, sélectionner les lignes en questions (toutes en l'occurrence), menu Données cliquer sur Tri et sélectionner la colonne selon laquelle le tri est souhaité.*

Quelles sont les deux lettres qui n'apparaissent pas dans cette fable ?

Réponse :

Combien de `"e"` apparaissent ?

Réponse :

▷ **Exercice 2 : Tracé de la carte de France.**

On utilise le fichier texte `Coordonnees_France.txt`. Comme on peut le constater en double-cliquant dessus sous Windows, ses lignes sont de la forme :

`Coordonnees_France.txt`

```
849460.0  6524534.0
848984.9  6525112.0
849022.0  6527350.0
...
```

Il s'agit des coordonnées des points frontières de la France métropolitaine. Ces coordonnées sont au format Lambert 93, système géodésique officiel depuis 2001.

a. Ouvrir depuis `Pyzo` le fichier `Coordonnees_France.txt`. Afficher toutes ses lignes.
C'est très long ! (CTRL-I pour stopper, ou la croix sur fond rouge au-dessus du Shell).

b. Compter le nombre de lignes de ce fichier.

Nombre de points :

c. Stocker les données contenues dans le fichier `Coordonnees_France.txt` dans deux listes `X` et `Y` de flottants, une pour la première colonne et l'autre pour la seconde.
Utiliser la méthode `split()` pour découper et la fonction `float` pour convertir en flottant.

```
>>> A="9.6"
>>> float(A)
```

d. Afficher le graphique reliant les points dont les coordonnées ont été stockées dans les listes `X` et `Y`. Tracer ce graphique en noir, dans un repère orthonormé.

**Compléments**

Les exercices suivants peuvent être traités dans le désordre.

▷ **Exercice 3 : îles.**

En réalité la France métropolitaine n'est pas un seul bloc, elle possède aussi des îles.

Les fichiers `Coordonnees_France_n.txt` où $n$ va de 1 à 28 contiennent toutes ces parties. Le premier contient la partie principale, les autres contiennent les îles.

Tracer à l'aide de ces fichiers la carte complète de France métropolitaine.

▷ **Exercice 4 : départements.**

Ouvrir sous Windows le fichier `Coordonnees_France_Dept.txt`.

Il contient la liste des points frontière des départements, chacun étant initialisé par une ligne `"Département n"`. Il contient aussi des lignes `"cut"` pour des coupures, permettant de tracer les îles.

Tracer la carte des départements métropolitains.

# Travaux Pratiques n°6
# Dictionnaires et applications

## I. Dictionnaires

En Python, le *dictionnaire* est un type construit, comme la liste. On accède à chaque élément (ou *valeur*) du dictionnaire grâce à un identifiant (ou *clé*). Par exemple, dans le dictionnaire `nombre_de_roues` défini par :

```
nombre_de_roues = {"voiture": 4, "vélo": 2, "tricycle": 3}
```

la valeur associée à la clé `"vélo"` est l'entier `2`. Les clés comme les valeurs peuvent être de différents types. À la différence d'une liste, un dictionnaire n'est pas une collection *ordonnée*.

▷ **Exercice 1 : Généralités.**

Taper les instructions suivantes dans l'éditeur de `Pyzo` :

**Définir un dictionnaire par ajout de clés et de valeurs**

```
dico_en_fr={}
dico_en_fr["yes"]="oui"
dico_en_fr["no"]="non"
dico_en_fr["why"]="pourquoi"
print(dico_en_fr)
```

L'instruction **in** permet de tester l'appartenance d'une clé à un dictionnaire, mais pas d'une valeur :

```
print("yes" in dico_en_fr)
print("pourquoi" in dico_en_fr)
```

**Supprimer des entrées**

```
del(dico_en_fr["no"])
print(dico_en_fr)
print(dico_en_fr.clear())
print(dico_en_fr)
```

**Autres façons de définir un dictionnaire**

```
    # définition directe
dico_es_fr={"si":"oui","hoy":"aujourd'hui","porque":"pourquoi"}
    # en utilisant la fonction dict
dico_en_nb=dict(one=1,two=2,three=3)
    # transformation d'une liste de tuple à 2 éléments
liste_es_nb=[("uno",1),("dos",2),("tres",3)]
dico_es_nb=dict(liste_es_nb)
    # définition par compréhension
suite_carre={x:x**2 for x in range(5)}
```

**Parcours d'un dictionnaire**

```
print(len(dico_es_fr))
for i in dico_es_fr.keys(): print(i)
print(dico_es_fr.values())
print(dico_es_fr.items())
```

# II. Applications

## A. Décodage

▷ **Exercice 2.**

Récupérer le fichier `morse.txt`. Vous y trouverez un dictionnaire `dict_morse` contenant l'alphabet Morse (inventé par Samuel Morse pour la télégraphie en 1832).

a. Écrire une fonction `trad_to_morse` prenant en argument un message sous forme de chaîne de caractères `message` (ne contenant que des majuscules sans accent et des espaces) et renvoyant une chaîne de caractères codant `message` en morse.

Pour séparer les mots de `message`, on utilisera la méthode `split`. Par défaut, le séparateur est une espace. Tester dans le Shell :

```
>>> message='HELLO WORLD'
>>> message.split()
>>> message.split('L')
```

Dans le message codé, chaque lettre doit être séparée d'une autre par une espace : `' '` et chaque mot d'un autre par trois espaces : `'   '`. On obtient par exemple :

```
>>> trad_to_morse('HELLO WORLD')
    '.... . .-.. .-.. ---    .-- --- .-. .-.. -..'
```

b. Écrire une fonction `inverser_dict` prenant en argument un dictionnaire et renvoyant un dictionnaire dont les clés sont les valeurs du dictionnaire initial et inversement. Appliquer cette fonction au dictionnaire `dict_morse` pour obtenir un dictionnaire de décodage du morse.

*Notez que cette fonction n'est utilisable que pour un dictionnaire* injectif*, c'est-à-dire dans lequel une même valeur n'est jamais associée à deux clés distinctes.*

c. À l'aide du dictionnaire de décodage, écrire une fonction de décodage du morse. Appliquer cette fonction à la chaîne `message_mystere`.

## B. Compter avec un dictionnaire

▷ **Exercice 3.**

Les dictionnaires peuvent également être utilisés comme *compteurs*. Compléter le programme suivant, qui crée un dictionnaire comptant le nombre d'apparitions de chaque lettre dans un mot :

```
mot = "abracadabra"
dico = dict()
for lettre in mot:
        if lettre in dico:
                dico[lettre] = ...
        else:
                dico[lettre] = ...
print(dico)
```

▷ **Exercice 4 : Analyse d'un fichier texte.**

Récupérer le fichier `LaFontaine.txt`.

Pour accéder à un fichier `txt` en Python sans avoir à en copier le contenu dans l'éditeur, on utilise la fonction **open**, en prenant garde à indiquer le chemin d'accès au fichier. En fin d'utilisation on le ferme grâce à la méthode `close`.

Un fichier peut être ouvert en lecture (**open** avec l'option `'r'`) ou en écriture (**open** avec l'option `'w'`). Cette option permet également de créer un fichier. Il existe d'autres options comme `'a'` (append) pour l'ajout à la fin du fichier.

Écriture dans un fichier

```
montxt=open('chemin\\fichier.txt'
   ,'w')
montxt.write('texte...')
montxt.close()
```

Lecture d'un fichier

```
montxt=open('chemin\\fichier.txt'
   ,'r')
s=montxt.read() # renvoie une
   chaîne de caractères
montxt.close()
```

a. Écrire une fonction `occurrences(a,s)` comptant le nombre d'occurrences d'un caractère `a` dans une chaîne `s`.

b. Ouvrir le fichier `LaFontaine.txt` à l'aide de Python. Stocker son contenu dans une chaîne de caractères. Convertir toutes ses majuscules en minuscules à l'aide de la méthode `lower`.

c. Définir un dictionnaire qui associe à chaque lettre de l'alphabet le nombre de fois où elle apparaît dans ce texte. Pour tester qu'un caractère c est une lettre, vous pouvez utiliser `c.isalpha()`, qui renvoie **True** si c est une lettre et **False** sinon.

d. Écrire une fonction `pourcentages` qui prend en argument un dictionnaire associant à chaque clé un nombre d'occurrences et renvoie un nouveau dictionnaire associant à chaque clé le pourcentage que représente les occurrences de cette clé sur le total des occurrences du dictionnaire. On pourra utiliser la fonction **round**`(x,n)`, qui arrondit le flottant `x` à la $n^{ème}$ décimale. Tester sur le dictionnaire précédent.

### C. Bases de données

À partir de plusieurs dictionnaires liés entre eux, il est possible d'en créer d'autres pour faire apparaître des informations souhaitées. C'est le principe des *bases de données*.

▷ **Exercice 5 : Noms de pays : le pays.**

Récupérer le fichier `countries.csv`.

a. L'accès à un fichier `csv` en Python suit à peu près le même principe que l'accès à un fichier `txt`. En implémentant les lignes suivantes, créer un dictionnaire des capitales :

```
import csv
fichier=open('chemin\\countries.csv','r')
lecture = csv.reader(fichier)
capitale = {ligne[1]:ligne[2] for ligne in lecture}

print(capitale['France'])
```

b. Créer, de la même façon, un dictionnaire `continent` associant les continents aux pays et un dictionnaire `monnaie` associant les monnaies aux pays.

c. Afficher les noms des pays d'Océanie, puis ceux dont la monnaie est l'euro.

d. Afficher les noms des pays non-européens dont la monnaie est l'euro.

e. Créer un dictionnaire associant à chaque continent la liste des monnaies qui y sont utilisées.

### D. Devinettes

▷ **Exercice 6 : Noms de pays : le nom.**

Reprendre le dictionnaire des capitales de l'exercice précédent.

a. L'ordinateur va choisir aléatoirement un nom de pays, et vous faire deviner sa capitale. En cas de succès, il vous répond `'Gagné !'`, sinon : `'Perdu ! La bonne réponse était...'`. On utilisera les commandes suivantes :

```
import random
random.choice(list(capitale.keys()))
```

b. Modifier le programme précédent pour qu'il compte 1 point à chaque succès, et affiche votre score final au bout de cinq parties.

c. Modifier le programme précédent pour qu'il vous autorise trois essais à chaque partie.

# Algorithmes gloutons

## I. Introduction

Les algorithmes gloutons sont utilisés dans des problèmes d'optimisation. Un problème d'optimisation consiste à déterminer le jeu de données d'entrée permettant de minimiser ou maximiser une fonction objectif, tout en satisfaisant à une ou des fonctions contraintes (il existe des problèmes avec ou sans contrainte).

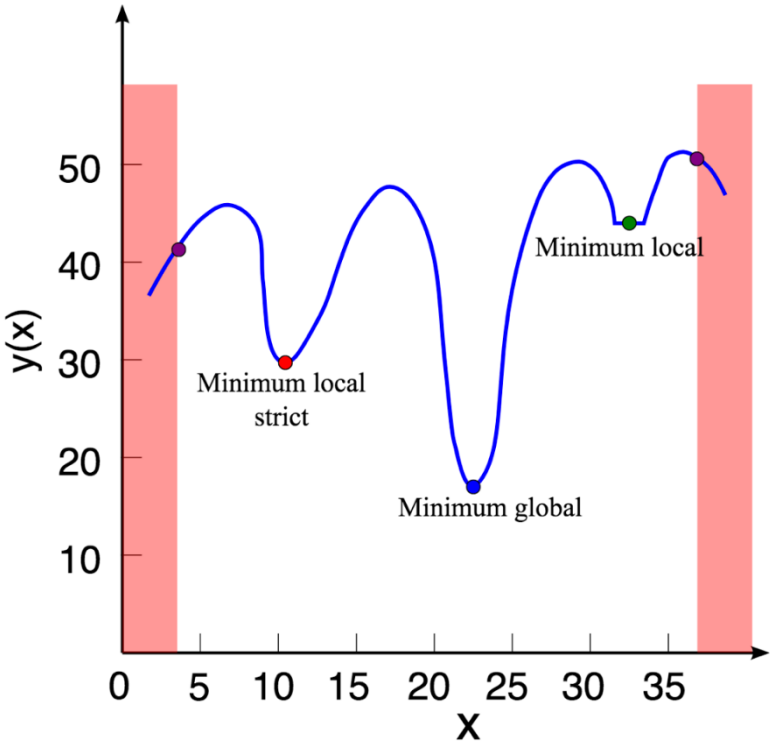
On peut citer par exemple le problème du choix d'itinéraire dans un réseau routier : il faut déterminer le trajet de sorte à minimiser le temps de parcours (ou la distance parcourue, ou la consommation, ou autre…), tout en respectant un ensemble de contraintes : il faut circuler sur les routes (on ne coupe pas à travers champs !) et respecter les sens de circulation.

Voici un second exemple :

- La recherche du minimum de la fonction $f : \begin{matrix} \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & x^2 - x - 2 \end{matrix}$ est un problème d'optimisation sans contrainte.
- La recherche du minimum de la fonction $f : \begin{matrix} \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & x^2 - x - 2 \end{matrix}$ sous la contrainte $x^3 \geq 8$ est un problème d'optimisation sous contrainte (contrainte d'inégalité). L'optimum est alors $x = 2$ et $f(2) = 0$. L'ensemble des solutions vérifiant les contraintes sont appelées **solutions valides** ou **valables**.

On distingue également plusieurs types d'optimum :

- optimum global ;
- optimum local strict (unique dans un intervalle réduit)
- optimum local (non unique dans un intervalle réduit)




Les algorithmes gloutons peuvent être adaptés aux problèmes d'optimisation pour lesquels il faut faire une série de choix pour arriver à une solution.

Certaines méthodes envisagent toutes les combinaisons possibles de ces choix afin de sélectionner la plus intéressante, mais cela peut être très couteux en mémoire et en temps de calcul.

Un algorithme glouton fait toujours le choix glouton : c'est le choix qui lui semble le meilleur sur le moment, en espérant que cela conduira à la solution optimale. Les méthodes gloutonnes sont bien plus simples car elles effectuent les choix sans se soucier de la suite de la résolution. Ces méthodes sont donc adaptées si elles permettent à coup sûr d'atteindre la solution optimale **globale** du problème.

## II. Un premier exemple : le problème du choix d'activité

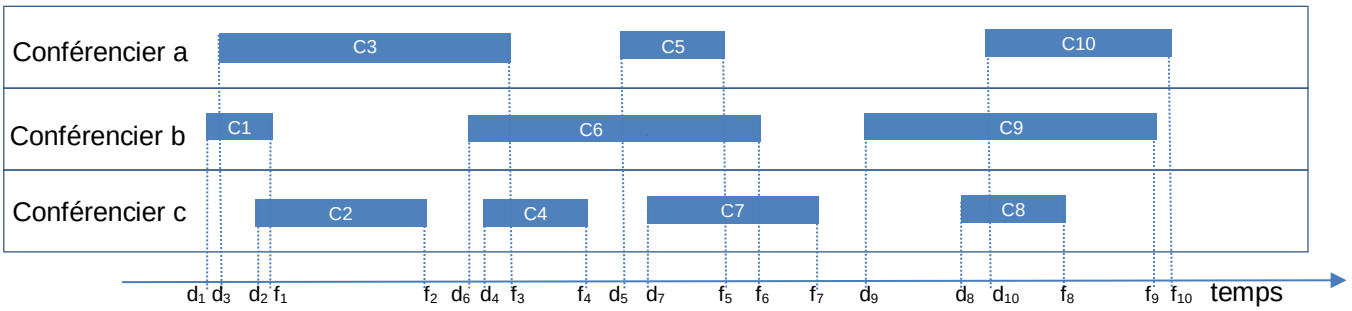
Dans cette partie, un premier exemple vous permettra une première approche de la méthode gloutonne. Suite à cela, un petit aperçu des fondements théoriques de la méthode vous sera exposé.

## 1. Présentation du problème

Le problème du choix d'activité est un problème dans lequel plusieurs activités souhaitent se voir attribuer une unique ressource.

Une application concrète est l'allocation d'une salle à différents conférenciers au cours d'une journée. Plusieurs personnes souhaitent effectuer une conférence, mais ces conférences ont lieu à des horaires définis. Certaines conférences démarrent alors que d'autres ne sont pas encore terminées. Il n'est donc pas possible d'attribuer la salle à chaque conférencier, et il faut faire des choix.

La solution optimale est celle qui permettra au plus grand nombre de conférences de se tenir.




Les conférences sont définies par leur heure de début et leur heure de fin. La conférence $c_i$ aura pour heure de début $d(c_i)$ et pour heure de fin $f(c_i)$.

Nous supposerons que les conférences ont été triées dans l'ordre croissant de leurs dates de fin. Si nous travaillons avec un ensemble C de n conférences, on a :

$$f(c_1) < f(c_2) < \cdots < f(c_{n-1}) < f(c_n)$$

On a bien affaire ici à un problème d'optimisation :
Si $S = \{s_1, s_2, \ldots, s_k\}$ est l'ensemble des conférences constituant la solution
Fonction objectif : maximiser le nombre de conférences : maximiser $|S|$
Fonction contrainte : les conférences doivent être compatibles, pour tout $i, j \in \{1, \ldots, k\}$, on doit avoir $f(s_i) \leq d(s_j)$ ou $f(s_j) \leq d(s_j)$

## 2. La solution gloutonne

- *Le principe du choix glouton*

La méthode gloutonne consiste à effectuer le choix qui parait le meilleur sur le moment, sans se soucier de la suite du problème. C'est le choix glouton.

Le choix glouton permet de déterminer la solution optimale pour un sous-ensemble du problème. Le reste du problème sera traité par des choix gloutons successifs, qui, associés entre eux, constituerons une solution optimale au problème.

- *Pour notre application*

Dans notre application, le premier choix glouton consiste à choisir la conférence qui termine le plus tôt possible, préservant ainsi la disponibilité de la salle pour les conférences ultérieures.

Lorsqu'un choix de conférence a été fait, on procède de la même manière pour le choix suivant, mais il y a la contrainte de compatibilité : il faut choisir une conférence compatible avec celles qui ont déjà été choisies : On ne peut choisir qu'une conférence qui débute après la fin de la dernière conférence choisie. Si on note $C$ l'ensemble des conférences, et $S$ l'ensemble des conférences qui constituent notre solution optimale, à l'itération $k$ on a :

$$S = \{s_1, s_2, \dots, s_k\} \text{ et } S \subset C$$

Alors il faut choisir la conférence $c_i \in C - S$ telle que $d(c_i) \geq f(s_k)$.

L'algorithme construit alors un ensemble $S$ de conférences mutuellement compatibles, c'est-à-dire tel que :

$$d(s_1) < f(s_1) \leq d(s_2 < f(s_2) \leq \cdots \leq d(s_j) < f(s_j)$$

## 3. Algorithme et test à la main

Paramètres : $C$ est la liste des conférences, représentées par leurs dates de début et de fin, et leur intitulé [d($c_i$), f($c_i$), '$c_i$']

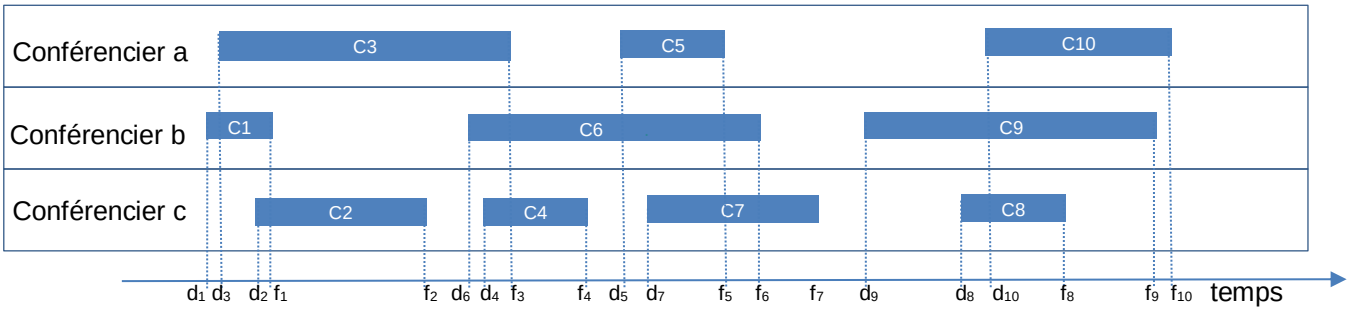
Algorithme :

#les conférences étant classées par ordre croissant de leur heure de fin, on sait que la
*Planning* <- [ *C*[0] ] # première valeur de $C$ correspond au premier choix glouton
*Fin* <- *C*[0][1]
Pour tous les éléments $c_i$ de la liste *C* :
    Si $c_i$ [0] >= Fin :
        Insérer $c_i$ à la suite de *Planning*
        Fin <- $c_i$ [1]

 1. Exécutez à la main cette méthode en utilisant l'ensemble de conférences illustré sur le schéma au bas de la page 1.

## 4. Implémentation

On modélise l'ensemble $C$ des conférences par une liste dont chaque élément modélise une conférence. Chaque conférence est modélisée par une liste de la date de début, la date de fin, et l'intitulé de la conférence. Cette liste est ordonnée par ordre croissant de la date de fin des conférences. Ainsi, l'exemple représenté sur l'illustration ci-dessous :




pourrait être modélisé par la liste :

L=[ [8,9,"C1"] , [8.45,10.3,"C2"] , [8.1,11.3,"C3"] , [11.15,11.45,"C4"] , [12,12.45,"C5"] , [11,13,"C6"] , [12.3,14,"C7"] , [16,17,"C8"] , [15,18,"C9"] ,[16.3,18.1,"C10"]]

 2. A la suite du script ***Algorithmes_gloutons.py*** implémentez la fonction ***choix_conferences( C )*** qui prend en argument une liste de conférences ***C***, et appliquant la méthode décrite précédemment.
Le résultat est une liste de conférences mutuellement compatibles, de longueur maximale.

 3. A la suite du script ***Algorithmes_gloutons.py*** tapez la ligne de code permettant d'utiliser la fonction ***choix_conferences( )*** pour déterminer la liste des conférences retenues parmi les conférences listées dans ***L***. Comparez le résultat avec celui obtenu en ***1.*** .

On observe que le résultat obtenu est bien un optimal. Cependant, il existe d'autres solutions, toutes autant optimales, par exemple C2, C4, C7, C10 ou encore C1, C4, C5, C9.

### 5. *Le choix glouton et le sous problème optimal*

Un algorithme glouton ne mène pas forcément à une solution optimale d'un problème d'optimisation.
Pour qu'un algorithme glouton soit adapté à la résolution d'un problème, il faut que le problème ait la propriété de *sous-structure optimale*, c'est-à-dire que la solution optimale du problème contient des solutions optimales des sous-problèmes.

Une résolution par la méthode gloutonne nécessite par ailleurs que les choix gloutons successifs engendrent une solution optimale globalement. C'est la propriété de choix glouton.

Pour notre application :

$S = \{s_1, s_2, \dots, s_k\}$ est le résultat de notre algorithme. On montre par récurrence que pour tout $j \leq k$, le sous-ensemble $S_j = \{s_1, s_2, \dots, s_j\}$ de $S$ est un sous ensemble d'une solution optimale :

- Pour $j = 0$, l'ensemble $S_j$ est un ensemble vide, et est donc un sous-ensemble d'une solution optimale
- Pour j>0 : Soit $T$ une solution optimale telle que pour $j < k$, $S_j \subset T$ ($T$ contient les activités $s_1, s_2, \dots, s_j$). On note b l'élément suivant $s_j$ dans T.
  - On a $f(s_{k+1}) \leq f(b)$ (en raison du choix glouton, on choisit l'activité que finit les plus tôt)
  - les éléments de T sont tous compatibles, ainsi chaque élément succédant à $b$ débute après la fin de $b$ et de $s_{k+1}$,

  Ainsi $T - \{b\} \cup \{a_{k+1}\}$ est une solution optimale, de même cardinal que $S$, et elle contient $a_1, a_2, \dots, a_{k+1}$

$S$ est donc une solution optimale.

On a bien démontré que l'algorithme du choix d'activité est correct, présente la propriété de sous-structure optimale, et que le choix glouton permet d'arriver à une solution optimale globalement.

## III. Un second exemple : le problème du sac à dos

### 6. *Présentation du problème*

Un promeneur souhaite transporter dans son sac à dos le fruit de sa cueillette. La cueillette est belle, mais trop lourde pour être entièrement transportée dans le sac à dos. Des choix doivent être faits. Il faut que la masse totale des fruits choisis ne dépasse pas la capacité maximale du sac à dos.

Les fruits cueilles ont des valeurs différentes, et le promeneur souhaite que son chargement soit de la plus grande valeur possible.

Image https://canifs.blogspot.com/2015/05/sacs-dos-vintage.html

| Fruits cueillis | Quantité ramassée | Prix au kilo |
|---|---|---|
| framboises | 1 kg | 24€/kg |
| Myrtilles | 3kg | 16€/kg |
| fraises | 5kg | 6€/kg |

La capacité du sac à dos n'est que de 5 kg.

On suppose que la masse d'un unique fruit est négligeable par rapport à la masse totale du sac en charge.

## 7. Présentation de l'algorithme et implémentation

Le principe pour optimiser le chargement est de commencer par mettre dans le sac la quantité maximale de fruits les plus chers par unité de masse S'il y a encore de la place dans le sac, on continue avec les fruits les plus chers par unité de masse parmi ceux restants, … et ainsi de suite jusqu'à ce que le sac soit plein. Il est possible qu'on ne puisse prendre qu'une fraction de la quantité de fruits disponible, si la capacité maximale du sac est atteinte. L'algorithme sera donc constitué d'une structure itérative, incluant une structure alternative.

Dans ***Algorithmes_gloutons.py***, on a ébauché l'algorithme de résolution du problème, et on a implémenté la cueillette par la liste :
cueillette = [ [24,1,"framboises"], [16,3,"myrtilles"], [6,5,"fraises"], [3,5,"mures"] ]

4. Dans ***Algorithmes_gloutons.py,*** compétez la fonction ***sac_a_dos( L , capacite )*** qui prend en argument une liste ***L*** modélisant la cueillette et la ***capacite*** du sac à dos, et appliquant la méthode décrite précédemment.
Le résultat est la liste des fruits contenus dans le sac à dos, correspondant au chargement de valeur maximale.

5. A la suite du script ***Algorithmes_gloutons.py*** tapez la ligne de code permettant d'utiliser la fonction ***sac_a_dos( )*** pour déterminer les quantités de fruits à choisir pour remplir le sac à dos, à partir de la liste ***cueillette.*** Exécutez votre programme et vérifiez le résultat.

L'annexe explique en quoi le problème ainsi formulé présente les propriétés de choix glouton et de sous-structure optimale. Une variante de ce problème ne trouve pas de solution optimale par la méthode gloutonne. Nous l'étudions ci-après.

## 8. Version non fractionnaire du problème du sac à dos

On suppose maintenant que les éléments à transporter ne sont pas fractionnables. Les fruits parmi lesquels choisir sont présentés ci-dessous :

| Fruits | Masse d'un fruit | Quantité disponible | Prix au kilo |
|---|---|---|---|
| Melon de cavaillon | 1 kg | 1 | 3 €/kg |
| Melon jaune | 2kg | 1 | 2.5 €/kg |
| Pastèque | 3kg | 1 | 2 €/kg |

Cet ensemble de fruits est modélisé dans ***Algorithmes_gloutons.py*** par la liste :
fruits_disponibles = [ [3,1,"melon de cavaillon"], [2.5,2,"melon jaune"], [2,3,"pastèque"] ]

L'objectif est toujours de placer dans le sac à dos le chargement de valeur maximale, de masse totale inférieure à 5kg. Par contre, les éléments n'étant fractionnables, il est possible que les choix successifs mène à un chargement qui ne remplit pas complètement le sac à dos.

On se propose de tester la méthode gloutonne pour cette nouvelle formulation.

 6. Dans ***Algorithmes_gloutons.py,*** copiez-collez et adaptez la fonction ***sac_a_dos( L , capacite )*** de sorte à ce qu'elle applique la méthode gloutonne à la version non fractionnaire du problème. Pour cela, il faut :
- commenter la structure conditionnelle permettant de continuer à remplir le sac si sa capacité restante est inférieure à la masse du i-ème fruit,
- adapter la condition de la structure itérative de sorte à ne continuer à remplir le sac que si la capacité le permet.
Exécutez la fonction en utilisant la liste ***fruits_disponibles***.

 7. Le résultat obtenu est-il optimal ? Comparer avec la solution { Pastèque , melon jaune}. Quelle est la solution dont la valeur est maximale ?

 8. De la propriété du choix glouton et de la sous-structure optimale, laquelle n'est pas respectée dans cette formulation du problème ? En quoi l'autre propriété l'est ?

Cette version du problème du sac à dos n'est pas adaptée à la méthode gloutonne. Le cours de deuxième année vous permettra d'aborder la programmation dynamique qui offre une solution optimale à ce problème : son principe consiste en l'étude de toutes les combinaisons possibles afin de ne retenir que la combinaison optimale.

## IV. Le problème du rendu de monnaie

Le problème du rendu de monnaie consiste en la détermination des pièces à utiliser pour rendre la monnaie.
Nous travaillons avec les valeurs entières du système monétaire européen : on note $S$ le n-uplet des valeurs entières $s_i$ du système monétaire européen

$$S = (500, 200, 100, 50, 20, 10, 5, 2, 1)$$

Nous disposons d'une quantité illimitée de pièces ou billets pour chacune des valeurs de $s$.

Si on note $x$ la valeur à rendre, on peut la représenter
par le n-uplet $K = (k_1, k_2, \dots, k_n)$ tel que $x = \sum_{i=1}^{n} k_i . s_i$.

Les $k_i$ représentent le nombre de pièces de valeur $s_i$. Par exemple, la valeur 63 sera représentée par le n-uplet $(0, 0, 0, 1, 0, 1, 0, 1, 1)$ car :

$$63 = (0 * 500 + 0 * 200 + 0 * 100 + 1 * 50 + 0 * 20 + 1 * 10 + 0 * 5 + 1 * 2 + 1 * 1)$$

La solution optimale est celle qui permet de rendre la monnaie en utilisant un minimum de pièces. On cherche donc à déterminer les valeurs des $k_i$ telles que la somme $\sum_{i=1}^{n} k_i$ soit minimale.

notons que les valeur de $S$ sont classées par ordre décroissant.

en partant de la plus grande valeur de $S$ :

L'algorithme glouton compare la valeur à rendre $v$ à la i-éme valeur de $S$ (noté $s_i$):

- si $s_i$ est supérieure à $v$, celle-ci ne pourra pas être utilisée dans le rendu de monnaie, alors on passe à la valeur suivante de $S$
- si $s_i$ est inférieur ou égale à $v$, alors on utilise $s_i$ dans le rendu de monnaie, et on poursuit le travail sur la nouvelle valeur à rendre $v - s_i$.
-

Ce processus est répété jusqu'à ce que la valeur à rendre $v$ soit nulle.

On ne connait pas à l'avance le nombre d'itérations nécessaires pour terminer le processus. Ainsi il sera nécessaire d'utiliser une structure itérative *tant que*.

 9. Compléter ci-dessous l'ébauche de l'algorithme de rendu de monnaie

```
v est la valeur à rendre
S=[500, 200, 100, 50, 20, 10, 5, 2, 1]
a_rendre = [ ]                          # liste des pièces ou billets à rendre
i=0
tant que v > 0 :
        si…… :          # si s_i est supérieure à v
                ……
        sinon :                         # si s_i est inférieur ou égale à v
                ……..# alors on utilise s_i dans le rendu de monnaie
                ……..# et on calcule le restant à rendre v − s_i.
```

 10. Testez l'algorithme en complétant le tableau ci-dessous, décrivant l'évolution des variables d'itération en itération pour une valeur à rendre de 14€. Vous remarquerez que les itérations 1 à 4 ont été éludées. En effet pour ces itérations là, les variables n'évoluent pas.

| $i$ | $v$ (en début d'itération) | $s_i$ | Restant de la valeur à rendre $v - s_i$ | Liste des piéces ou billets à rendre |
|---|---|---|---|---|
| 0 | 14 | 500 | Sans objet | [ ] |
| 1 | 14 | 200 | Sans objet | [ ] |
| … | … | … | … | … |
| 5 | 14 | 10 | 4 | [10 ] |
| 6 | 4 | 5 | Sans objet | [10 ] |
| 7 | 4 | 2 | | |
| | | | | |
| | | | | |

 11. A la suite du script ***Algorithmes_gloutons.py*** implémentez la fonction ***rendu_monnaie( valeur , systeme )*** qui prend en argument la ***valeur*** à rendre et la liste ***systeme*** des valeurs du système monétaire utilisé, et appliquant la méthode décrite précédemment.
Le résultat est une liste des pièces et billets utilisés pour le rendu de monnaie.

 12. A la suite du script ***Algorithmes_gloutons.py*** tapez la ligne de code utilisant la fonction définie précédemment.

Un algorithme glouton ne mène pas toujours à une solution optimale. C'est aussi vrai pour l'algorithme de rendu de monnaie. Nous avons obtenu dans les questions précédentes une solution optimale parce que le système monétaire utilisé était adapté. On dit que le système est canonique. Pour d'autres systèmes monétaires, la méthode n'apporte pas une solution optimale.

 13. Exécutez la fonction ***rendu_monnaie( )*** pour déterminer les pièces à utiliser pour rendre 14€ dans le système monétaire (9,7,3,1). Le résultat obtenu est-il optimal ?

Il est difficile de définir un système canonique, par contre, il existe des méthodes permettant de tester si un système existant est canonique.

De manière générale, il est difficile de déterminer si un algorithme glouton amènera à une solution optimale pour un problème donné. Une théorie basée sur les « matroïdes » permet d'identifier des problèmes pouvant être résolus de manière optimale par une méthode gloutonne, mais certains problèmes non identifiés par cette théorie sont toutefois adaptés à la résolution par un algorithmes glouton.

## V. Variante du problème de choix d'activité

Le problème du choix d'activité peut être résolu en utilisant une variante de la méthode proposée en début de sujet : au lieu de choisir la première activité à terminer, on peut choisir la dernière activité à débuter.

 14. Comment mettre en forme les données pour appliquer la méthode gloutonne ainsi formulée ?

Le script ***Algorithmes_gloutons.py*** fournit la liste des activité en respectant cette mise en forme.

 15. Construire un programme appliquant cette variante de la méthode de résolution du problème de choix d'activité. Le résultat obtenu est-il le même que celui obtenu en début de TP ? Le résultat obtenu est-il optimal ?

## VI. Annexe – Le choix glouton et le sous problème optimal pour le problème du sac à dos

Le problème du sac à dos présente bien la propriété de choix glouton : Soit $C$ l'ensemble des fruits de notre cueillette, et soit S l'ensemble correspondant à une solution optimale. A l'itération k, l'ensemble $S$ est en cours de composition, et on a déjà déterminé l'ensemble $S_k = \{s_1, s_2, \dots, s_{k-1}\}$ . L'ensemble $C$ a été ôté des fruits $s_1, s_2, \dots, s_{k-1}$, et il reste l'ensemble $C_k = C - S_k$. Le choix réalisé à l'itération $k$ est fait dans cet ensemble et si le sac n'est pas plein, on choisi l'élément $s_k$ de plus forte valeur massique. C'est ce choix, localement optimal, qui permettra d'obtenir globalement une solution optimale. En effet, si on ne choisit pas cet élément $s_k$, on choisira alors un élément de valeur massique inférieure, et à la fin de cette itération la

valeur du sac ne sac pas maximale. Si les itérations suivantes ne nous permettent pas d'insérer cet élément $s_k$, alors le résultat obtenu ne correspondra pas au chargement de valeur maximale. Le choix localement optimal amènera donc à une solution optimale globalement. La solution proposée présente donc bien la propriété du choix glouton.

Le problème du sac à dos présente également la propriété de sous-structure optimale : à l'itération $k$, la solution $S$ est en cours d'élaboration, et l'algorithme a déjà déterminé le sous-ensemble $S_k$ de $S$. Il reste à déterminer le reste $S_{\text{n-k}}$ de la solution optimale $S$. On peut écrire $S = S_k \cup S_{\text{n-k}}$ . L'ensemble $S_k$ a été constitué de sorte à obtenir un ensemble de $k-1$ éléments, de valeur maximale. Il reste à choisir parmi les éléments de $C_k = C - S_k$. $C_k$ est le sous-problème. La solution $S$ sera optimale si $S_{\text{n-k}}$ est la solution optimale du sous-problème $C_k$ : en effet à l'itération k on a déjà choisi les $k-1$ éléments de valeur maximale, et il reste à choisi parmi les éléments restants de notre cueillette. La valeur de notre sac sera maximale si on choisit parmi ces éléments le sous-ensemble de valeur maximale. Le problème présente donc bien la propriété de sous-structure optimale.

Références :
Algorithmique – Cormen Leiserson Rivest Stein - Dunod
Matroïdes et algorithmes gloutons : UNE INTRODUCTION - Pierre Béjian
Algorithmes gloutons - Ressource éduscol NSI

# Travaux Pratiques n°8
# Résolution d'équations

Le but de cette séance est de programmer et de tester des algorithmes de résolution numérique d'équations de la forme $f(x) = 0$.

## I. Introduction

▷ Créer un répertoire `TP08`.

▷ Récupérer le programme `ModeleGraphique.py` dans le répertoire `TP04` ou sur le site prepabellevue.org. Ce programme permet de tracer des courbes grâce au module matplotlib.

**Rappel.** Pour définir une fonction numérique on peut privilégier la méthode rapide :

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

plutôt que

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

## II. Dichotomie

On suppose donnée une fonction $f$ continue sur un intervalle $[a, b]$, telle que $f(a)$ et $f(b)$ sont de signes opposés. Il existe alors $c \in [a, b]$ tel que $f(c) = 0$, d'après le théorème des valeurs intermédiaires.

L'algorithme de dichotomie permet de calculer une approximation d'un tel réel $c$ aussi proche que souhaitée. Il consiste, à chaque étape, à partager l'intervalle d'étude en deux moitiés, et à identifier la moitié où se trouve $c$.

Compléter le pseudo-code ci-dessous :

**Fonction dichotomie**
- **Données :** $f$ (fonction), $a$, $b$ (réels), $\varepsilon$ (réel, par défaut égal à $2^{-51}$)
- $x \leftarrow a$
- $y \leftarrow b$
- **Tant que** $|y - x| > \varepsilon$ **faire**
  - $z \leftarrow$
  - **Si** $f(x)$ et $f(z)$ sont de même signe **alors**
    - 
  - **sinon**
    - 
  - **fin du Si**
- **fin du Tant que**
- **Résultat :**

**fin de la fonction.**

▷ **Exercice 1.**

a. Compléter l'algorithme de dichotomie et le programmer en Python.

b. Prendre soin dans cette question de nommer les fonctions différemment, par exemple $f_1$, $f_2$, $f_3$, car elles seront réutilisées dans la suite.
Tester la fonction de dichotomie pour le calcul de :

*(i)* $\sqrt{2}$, qui est solution de l'équation $x^2 - 2 = 0$ sur l'intervalle $[1, 2]$,

$\sqrt{2} =$

*(ii)* la racine 7-ème de 1000 (Comment vérifier ?)

$\sqrt[7]{1000} =$

*(iii)* les racines réelles de $P = 4X^4 - 2X^3 + X^2 - 18X + 8$.
Tracer auparavant la courbe représentative de $x \mapsto P(x)$ pour connaître leur nombre et les encadrer.

Racines :

c. Modifier la fonction de dichotomie de façon à ce qu'elle renvoie également (en seconde variable de sortie) le nombre d'itérations effectuées pour calculer la solution.
Noter ce nombre pour les exemples ci-dessus.

Nombres d'itérations :

## III. Méthode de Newton

Si $f$ est une fonction dérivable sur un intervalle $[a, b]$, et $\alpha$ est une solution de l'équation $f(x) = 0$ dans cet intervalle, alors la méthode de Newton construit une suite qui, si elle converge, converge vers $\alpha$, et ceci de façon très rapide.

**Méthode de Newton**

On choisit une valeur judicieuse de $u_0$ et on pose :

$$\forall n \in \mathbb{N} \qquad u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$$

Attention : il n'est pas garanti que la suite $(u_n)$ ainsi construite est bien définie, ni qu'elle converge.

Il est parfois possible de calculer $h(x) = x - \frac{f(x)}{f'(x)}$, et de définir plutôt $u_{n+1} = h(u_n)$.

▷ **Exercice 2. Programmation et tests.**

a. Écrire une fonction de calcul de $\alpha$ par la méthode de Newton. Cette fonction recevra les fonctions $f$ et $f'$, le réel $u_0$ et l'erreur tolérée (argument optionnel égal à $2^{-51}$).

b. Tester la fonction sur les exemples de l'exercice précédent.

c. Ajouter en variable de sortie le nombre d'itérations effectuées.
Comparer ces nombres avec ceux de l'exercice précédent.

Nombres d'itérations :

▷ **Exercice 3. Calcul de la dérivée.**

En pratique, il peut arriver que le calcul de la dérivée de $f$ soit difficile voire impossible.

On peut calculer numériquement sa dérivée. L'algorithme de Newton pourra alors être reprogrammé sans la variable d'entrée $f'$.

Une méthode efficace est d'utiliser l'approximation : $f'(x) \simeq \dfrac{f(x+h)-f(x-h)}{2h}$

avec $h$ petit. Le choix de $h$ est important, il ne doit pas être trop petit. On admet que $h = 2^{-17}$ donne de bons résultats.

On rappelle que les fonctions usuelles sont définies par le module `numpy`.

a. Créer une fonction `Der(f,x,h)` qui calcule une approximation de $f'(x)$. La variable $h$ recevra une valeur optionnelle.

b. Créer un graphique représentant sur l'intervalle $[-5, 2]$ les courbes de l'exponentielle et de sa dérivée approchée par la fonction `Der`.
Déterminer le maximum de la différence entre ces deux courbes.

Différence maximale :

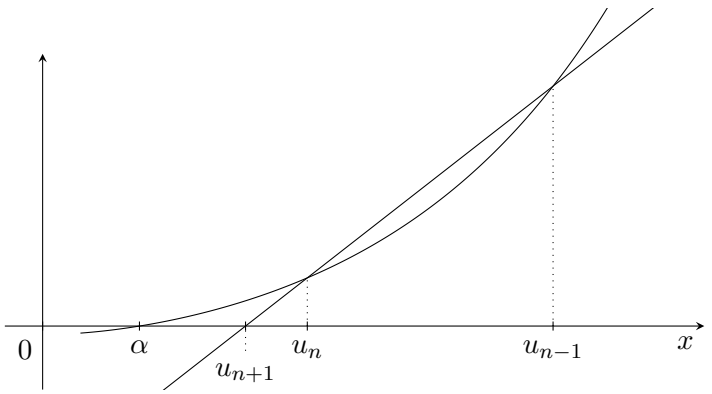
c. Reproduire la question précédente avec la fonction sinus sur l'intervalle $[0, 10]$.

Différence maximale :

▷ **Exercice 4. Méthode des sécantes.**

Cette méthode évite le calcul ou l'approximation de la dérivée de $f$.

Même si on dégrade légèrement la vitesse de convergence de la méthode, on utilise plutôt la corde reliant les points $(u_{n-1}, f(u_{n-1}))$ et $(u_n, f(u_n))$ : $u_{n+1}$ est l'abscisse de l'intersection de cette corde avec l'axe $(Ox)$.




a. Écrire une fonction qui prend comme paramètres la fonction $f$, les valeurs de départ $u_0$ et $u_1$, une précision optionnelle $\varepsilon$, et qui renvoie la valeur de la limite à $\varepsilon$ près.

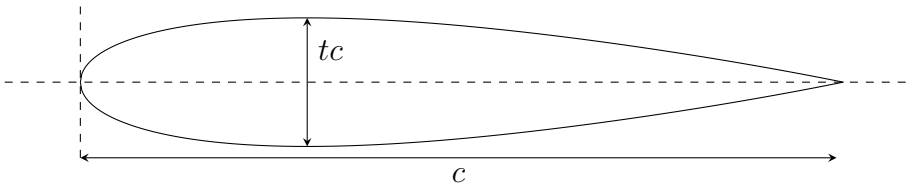
b. Tester cette fonction avec les exemples précédents.

c. Ajouter en variable de sortie le nombre d'itérations effectuées.
Comparer ces nombres avec ceux des exercices précédents.

Nombres d'itérations :

# IV. Applications

▷ **Exercice 5. Aile d'avion.**




Les profils NACA sont des profils aérodynamiques pour ailes d'avions, conçus par la NACA (National Advisory Committee for Aeronautics soit Comité Consultatif National pour l'Aéronautique), prédécesseur de la NASA.

Le profil symétrique utilise la fonction :

$$f(x) = tc\left[1{,}4845\sqrt{\frac{x}{c}} - 0{,}63\frac{x}{c} - 1{,}758\left(\frac{x}{c}\right)^2 + 1{,}4215\left(\frac{x}{c}\right)^3 - 0{,}5075\left(\frac{x}{c}\right)^4\right]$$

Cette fonction donne la demi-épaisseur en fonction de $x \in [0,1]$, où $c$ est la longueur de la corde et $t$ est l'épaisseur maximale en tant que fraction de $c$.

Nous supposerons que $c = 1$ et $t = 0{,}17$. On obtient le profil NACA0017, visible ci-dessus.

a. Tracer ce profil.

b. En quel point $x$ l'épaisseur est-elle maximale ? Quelle est alors cette épaisseur ?

Réponses :

▷ **Exercice 6. Vérification de la stabilité d'un vérin de pelle hydraulique**

Dans cet exercice, on se propose de vérifier que la commande automatique de la force dans le vérin est stable.

On parle de stabilité lorsque pour une consigne d'effort donnée bornée, l'effort réel transmis par le vérin est lui-même borné.

Pour cela, il faut proposer un modèle du système. On utilise la fonction de transfert du système (rapport des grandeurs de sortie et d'entrée du système), qui s'exprime pour le vérin de la manière suivante :

$$\frac{F(p)}{F_c(p)} = \frac{K}{p^4 + ap^3 + bp^2 + cp + d}$$

Avec $K = 1$ $a = -41{,}9$ $b = -359{,}4$ $c = 15\,211$ $d = 141\,150$,

où $F$ représente l'effort réel dans le vérin, et $F_c$ l'effort de consigne. Pour une question de simplicité en automatique des systèmes, on préfère travailler avec la variable de Laplace $p$ plutôt que la variable temporelle $t$. On rappelle la relation (si les hypothèses d'existence de l'intégrale sont validées) entre une fonction temporelle et sa transformée dans le domaine de Laplace :

$$F(p) = \int_0^{+\infty} e^{-pt}\, dt$$

$F$ est donc la transformée de Laplace de la fonction $f$.

On précise que la commande du vérin est stable à condition que toutes les racines du dénominateur de sa fonction de transfert soient négatives.

À l'aide de l'algorithme de dichotomie, préciser si le système est stable ou instable.

Réponse :

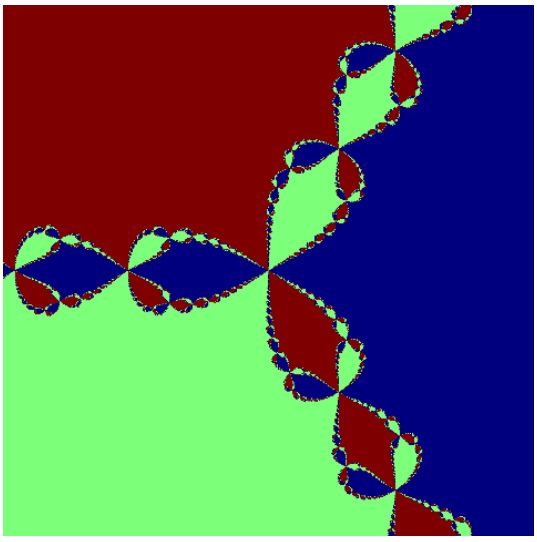
▷ **Exercice 7. La fractale de Newton.**

On admet que pour $f(x) = x^3 - 1$ et $u_0$ complexe alors la suite définie par la méthode de Newton converge dès qu'elle est définie.

Sa limite est une des solutions de l'équation $f(x) = 0$, donc 1 ou $j = e^{i\frac{2\pi}{3}}$ ou $j^2 = e^{i\frac{4\pi}{3}}$.

Ceci définit trois zones du plan complexe $C(1)$, $C(j)$ et $C(j^2)$. Un complexe $z$ est dans la zone $C(\zeta)$ si la suite débutant en $u_0 = z$ converge vers $\zeta$.

Ces trois zones sont appelées bassins d'attraction. Leurs frontières forment une fractale.



**Les complexes en Python**

Définition du complexe $2+4i$ : `z=complex(2,4)`

Partie réelle, partie imaginaire, module : `z.real` `z.imag` `abs(z)`

a. Écrire une fonction qui à un complexe $z$ associe 0, 1 ou 2 selon que la limite de la suite définie par la méthode de Newton soit 1, $j$ ou $j^2$.

b. Importer le module numpy et la fonction `imshow` du module matplotlib.pyplot.

La fonction `imshow` reçoit une matrice numpy d'entiers, et attribue une couleur à chaque entier. Elle trace l'image représentée par la matrice.

Tester la fonction avec

```
A=array([[0,1,2],[2,1,1]])
imshow(A,interpolation="nearest")
```

puis

```
B=[arange(10)]
imshow(B,interpolation="nearest")
```

L'option `interpolation="nearest"` sera inutile dans la suite.

c. Tracer les bassins d'attraction. Pour ceci

- Définir quatre réels $a$, $b$, $c$, $d$ qui déterminent la zone affichée $[a,b]\times[c,d]$. Poser initialement $(a,b)=(c,d)=(-1{,}5;1{,}5)$.
- Définir le nombre de points $N=200$. Il sera remplacé par $N=1000$ une fois le programme terminé.
- Créer à l'aide de l'instruction `linspace` du module numpy le tableau des abscisses $X$ ($N$ points équirépartis de $a$ à $b$) et le tableau des ordonnées $Y$ ($N$ points équirépartis de $c$ à $d$).
- Créer une matrice nulle de taille $(N,N)$ du type `array` de numpy.
- Affecter tous les coefficients de cette matrice, le coefficient $(i,j)$ recevant la valeur 0, 1 ou 2 selon la limite de la suite définie par $u_0=z$ où $z$ est le complexe défini par **complex**`(X[i],Y[j])`. Utiliser la fonction de la question a.
- Afficher cette matrice à l'aide de l'instruction `imshow`.

On pourra utiliser la méthode de Newton avec une petite précision, comme $\varepsilon=2^{-2}$.

d. Afficher quelques agrandissements. Par exemple la fenêtre $[0;0{,}5]\times[0;0{,}5]$, puis la fenêtre $[0{,}3;0{,}4]\times[0{,}2;0{,}3]$, etc.

# Travaux Pratiques n°9
# Algorithmes récursifs

Fondamentale en mathématiques, la *récursivité* consiste à décrire un objet (concept, processus, structure...) en faisant appel à ce même objet. En informatique, son emploi a été permis par le développement des langages de programmation Lisp et ALGOL 60 à la fin des années 1950.

Nous allons implémenter au cours de ce TP différents *algorithmes récursifs*.

## I. Principe

Un algorithme est dit *récursif* s'il s'appelle lui-même, contrairement à un algorithme *itératif*.

▷ **Exercice 1 : Un premier exemple : calcul de factorielle.**

Rappelons la fonction itérative permettant de calculer $n!$, rencontrée lors du TP n°3 :

```
def facto_it(n):
    f = 1
    for i in range(1, n+1):
        f *= i
    return f
```

Dans cette fonction, le résultat est calculé par itérations successives : on calcule 1!, puis 2!, 3!, ..., $(n-1)!$, et enfin $n!$.

L'équivalent récursif de cette fonction consiste à calculer $(n-1)!$ en faisant appel à la fonction elle-même :

```
def facto_rec(n):
    if n=0:
        return 1
    else:
        return n*facto_rec(n-1)
```

a. Écrire et tester ces deux fonctions.

b. Dans l'algorithme récursif, pourquoi est-il important de traiter le cas $n = 0$ à part ? On parle de *cas trivial*.

c. Comparer le nombre d'itérations de la première fonction au nombre d'appels de la seconde. Qu'en conclure sur le temps de calcul ?

▷ **Exercice 2 : Exponentiation rapide.**

Étant donnés un réel $x$ et un entier naturel $n$, on veut écrire un algorithme renvoyant $x^n$ (sans utiliser `x**n`). Dans un premier temps, on utilise la définition (récursive !) : $x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^{n-1} & \text{sinon} \end{cases}$.

a. Écrire deux fonctions, l'une itérative, l'autre récursive, prenant en argument $x$ et $n$ et renvoyant $x^n$. Comparer le nombre d'itérations de la première fonction au nombre d'appels de la seconde.

b. On fait à présent appel à l'*exponentiation rapide* : en utilisant le fait que :

$$x^n = \begin{cases} (x^{n//2})^2 & \text{si } n \text{ est pair} \\ x \times (x^{n//2})^2 & \text{si } n \text{ est impair} \end{cases},$$

écrire une fonction récursive renvoyant $x^n$. Quel est le cas trivial ?

c. Donner l'ordre de grandeur du nombre d'appels de cette fonction.

▷ **Exercice 3 : Limite : le cas de la suite de Fibonacci.**

Certains problèmes sont plus ou moins bien adaptés à une résolution récursive. On rappelle la définition de la suite de Fibonacci $(F_n)$ :

$$\begin{cases} F_0 = 0,\ F_1 = 1 \\ \forall n \geq 0,\ F_{n+2} = F_{n+1} + F_n \end{cases}.$$

a. Écrire une fonction récursive `fibo(n)` codant cette définition et renvoyant $F_n$.

b. Calculer `fibo(10)`, `fibo(20)`, `fibo(30)`... Que constate-t-on ? Expliquer en estimant le nombre d'appels.

c. Écrire une fonction itérative prenant en argument $n$ et renvoyant $F_n$. Combien y a-t-il d'itérations ? Comparer avec la version récursive.

# II. Applications

▷ **Exercice 4 : Recherche dichotomique.**

Nous avons implémenté lors du TP n°6 un algorithme itératif de recherche dichotomique d'un nombre donné $x$ dans une liste $L$ **triée** (par ordre croissant) de $n$ nombres. Pour rappel :

1. on pose $a = 0$ et $b = n - 1$,
2. on pose $i = (a + b)//2$,
3. $\begin{cases} \text{si } L[i] = x, & \text{c'est terminé} \\ \text{si } L[i] > x, & \text{on pose } b = i - 1 \\ \text{si } L[i] < x, & \text{on pose } a = i + 1 \end{cases}$,
4. on réitère 2. et 3. jusqu'à ce que $a > b$.

a. Reformuler les étapes 3. et 4. de façon récursive.

b. Quel est le cas trivial de cet algorithme récursif ?

c. Écrire une fonction récursive `dicho(x,L)` de recherche dichotomique de $x$ dans une liste triée $L$. Tester cette fonction.

▷ **Exercice 5 : Permutations.** Une *permutation* d'un ensemble fini est un arrangement de ses éléments. Par exemple, les permutations de $a$, $b$ et $c$ sont :

$$abc,\ acb,\ bac,\ bca,\ cab,\ cba.$$

On représente un tel ensemble par une chaîne de caractères $s$ (dans l'exemple, `'abc'`) de longueur $n$, et on veut obtenir l'ensemble de ses permutations sous forme de liste $L$.

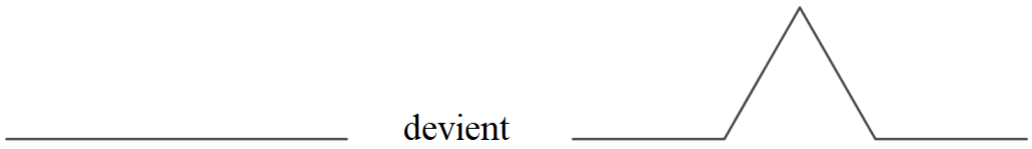
a. Quelle sera la taille de la liste $L$ ?

b. On met à part le premier élément de $s$. En supposant connues toutes les permutations des autres éléments, comment obtenir toutes les permutations de $s$ ? En déduire un algorithme récursif permettant de déterminer $L$.

c. Écrire une fonction récursive `permutations(s)` mettant en œuvre cet algorithme. Tester cette fonction.

▷ **Exercice 6 : Le flocon de Koch.**

Le *flocon de Koch* est une image fractale construite par récurrence. On applique à chaque côté d'un triangle équilatéral la transformation suivante :




puis on applique la même transformation à la figure obtenue, et ainsi de suite.

Pour tracer cette figure en Python, on va utiliser le module `turtle`. Ce module permet de simuler le déplacement de la pointe d'un stylo sur le plan. Celle-ci est initialement située à l'origine (0,0). Pour tracer un segment, on utilise les fonctions suivantes, à tester successivement dans le Shell :

```
>>> import turtle as t
>>> t.reset()
>>> print(t.pos())
>>> t.goto(100,100)
>>> print(t.pos())
>>> t.goto(300,-100)
>>> t.goto(400,0)
>>> t.exitonclick()
```

L'image obtenue au bout de $n$ transformations est donnée par la fonction :

```
import turtle as t
import math as m

def flocon(n):
    try:
        t.reset()    #réinitialisation de la figure
    except t.Terminator:    #permet d'éviter une exception bloquante
        pass
    cote(200,m.pi/3,n)     #tracé des cotés du triangle
    cote(200,-m.pi/3,n)
    cote(200,-m.pi,n)
    t.exitonclick()
```

où la fonction `cote(l,a,n)`, qui est utilisée pour tracer successivement les trois côtés de la figure, est une fonction récursive :

- si $n = 0$, on trace un simple segment de longueur $l$ et d'angle $a$ (par rapport à $[Ox)$)
- si $n > 0$, on fait appel quatre fois à la fonction `cote` sous la forme `cote(l_k,a_k,n-1)` où, pour tout $k \in [\![1,4]\!]$, $l_k$ et $a_k$ sont les longueurs et angles respectifs des segments obtenus par transformation du segment de longueur $l$ et d'angle $a$.

a. Écrire la fonction `cote`.
b. Tester la fonction `flocon` pour différentes valeurs de $n$.

▷ **Exercice 7 : Les tours de Hanoï.**

Le *jeu des tours de Hanoï* consiste à déplacer des disques de différents diamètres d'une pile (appelée *tour*) de départ à une pile d'arrivée en passant par une pile intermédiaire, sachant que :

- on ne peut déplacer qu'un disque à la fois,
- un disque ne peut pas être placé sur un disque plus petit que lui.

On représente les disques par leurs diamètres, et les tours par des listes. Avec cinq disques, les configurations de départ et d'arrivée sont les suivantes :

```
# Configuration de départ
A,B,C=[5,4,3,2,1], [], []

# Configuration d'arrivée
A,B,C=[], [], [5,4,3,2,1]
```

L'algorithme récursif de résolution de ce jeu est particulièrement élégant : pour déplacer $n$ disques de $A$ vers $C$ en passant par $B$ :

1. déplacer les $(n-1)$ plus petits disques de $A$ vers $B$ en passant par $C$,
2. déplacer le plus grand disque de $A$ vers $C$ en passant par $B$,
3. déplacer les $(n-1)$ plus petits disques de $B$ vers $C$ en passant par $A$.

a. Écrire une fonction récursive `hanoi(n)` permettant de résoudre le jeu à $n$ disques. On affichera la configuration des tours à chaque étape.

b. Tester la fonction `hanoi` pour différentes valeurs de $n$.

# Travaux Pratiques n°10
# Fractales

## I. Systèmes de fonctions itérées

▷ **Exercice 1 : Jeu du chaos.**

Le *jeu du chaos* consiste, à partir d'un triangle équilatéral $ABC$ et d'un point $M_0$ choisi au hasard à l'intérieur de ce triangle, à construire pour tout $i$, récursivement, le milieu $M_{i+1}$ de $M_i$ et d'un sommet $S$ au hasard (parmi $A$, $B$ et $C$ donc).

a. Implémenter cet algorithme, de la façon suivante :
   - Définir les coordonnées de $A$, $B$ et $C$,
   - Définir le point $M_0$ comme le point d'affixe $z = \frac{pa+qb+rc}{p+q+r}$ où $a$, $b$, $c$ sont les affixes respectives de $A$, $B$ et $C$, et $p$, $q$, $r$ sont des réels de ]0,1[ choisis aléatoirement à l'aide de la fonction `random` du module `random`,
   - Définir $N = 100$. Pour tout $i$ de 1 à $N$, choisir un sommet $S$ à l'aide de la fonction `randint` du module `random`, et définir le milieu $M_{i+1}$ de $M_i$ et de $S$,
   - Représenter les points $A$, $B$, $C$ et $M_i$ à l'aide de la fonction `scatter` du module `matplotlib.pyplot`. L'option $s$ permet de modifier la taille des points.

   Augmenter la valeur de $N$. Quelle figure obtient-on ?

b. Procéder de même en remplaçant le triangle initial par un pentagone régulier.

▷ **Exercice 2 : La fougère de Barnsley.**

Un *système de fonctions itérées* est un ensemble de fonctions contractantes $f_i : \mathbb{R}^2 \to \mathbb{R}^2$, où $i$ parcourt un ensemble $I$.
La fonction $f = \bigcup_{i\in I} f_i : (\mathbb{R}^2)^I \to (\mathbb{R}^2)^I$ associe alors à toute famille de points de $\mathbb{R}^2$ $(x_i)_{i\in I}$ la famille de points de $\mathbb{R}^2$ $(f_i(x_i))_{i\in I}$. Cette application est également contractante et admet donc, d'après le théorème du point fixe, un et un seul *attracteur*, c'est-à-dire une partie de $\mathbb{R}^2$ fixe par $f$. On admet ce résultat ici.
Les figures obtenues dans l'exercice précédent sont des exemples de tels attracteurs.
En voici un autre : on définit la suite $(X_n) = \begin{pmatrix} x_n \\ y_n \end{pmatrix}$ de points de $\mathbb{R}^2$ par :
$x_0 = y_0 = 0$ et : $\forall n \in \mathbb{N}$, $X_{n+1} = AX_n + B$,
où $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ et $B = \begin{pmatrix} g \\ h \end{pmatrix}$ avec une probabilité $p$, avec :

| $p$ | $a$ | $b$ | $c$ | $d$ | $g$ | $h$ |
|---|---|---|---|---|---|---|
| 0.01 | 0 | 0 | 0 | 0.16 | 0 | 0 |
| 0.85 | 0.85 | 0.04 | −0.04 | 0.85 | 0 | 1.60 |
| 0.07 | 0.2 | −0.26 | 0.23 | 0.22 | 0 | 1.60 |
| 0.07 | −0.15 | 0.28 | 0.26 | 0.24 | 0 | 0.44 |

Implémenter cet algorithme, et représenter les points $X_n$. Quelle figure obtient-on ?

## II. Les ensembles de Julia et de Mandelbrot

▷ **Exercice 3.**

Étant donné un nombre complexe $c$, on définit la suite complexe $(z_n)$ par :

$$z_0 \in \mathbb{C} \quad \text{et} \quad \forall n \geq 0,\ z_{n+1} = z_n^2 + c.$$

L'*ensemble de Julia* associé à $c$ est l'ensemble des nombres complexes $z_0$ pour lesquels la suite $(z_n)$ est bornée.
En Python, le nombre complexe $a + ib$ est défini par `a+b*1j` ou par **complex**`(a,b)`. Son module est donné par la fonction **abs**, et ses parties réelle et imaginaire sont respectivement données par les fonctions `real` et `imag` du module `numpy`.

a. Définir $c = 0$. Écrire une fonction `julia(x,y)` qui renvoie **True** si la suite $(z_n)$ associée à $z_0 = x + iy$ est bornée, **False** sinon. On admettra que la suite $(z_n)$ est bornée si le module de $z_n$ reste inférieur à 2 jusqu'au rang $N = 20$.

b. Voici quelques valeurs intéressantes de $c$ :

$$0{,}3 + 0{,}5i, \quad 0{,}285 + 0{,}013i, \quad -0{,}8 + 0{,}156i, \quad -0{,}4 + 0{,}6i, \quad -0{,}123 + 0{,}745i.$$

On admet qu'un ensemble de Julia est toujours contenu dans le rectangle $[-2,2]^2$.

- Choisir une valeur de $c$,
- À l'aide de la fonction `linspace` de `numpy`, parcourir le rectangle $[-2,2]^2$ avec un pas $p = 0.1$ en stockant les points qui font partie de l'ensemble de Julia,
- Représenter l'ensemble de Julia à l'aide de la fonction `scatter`. L'option $s$ de cette fonction permet de modifier la taille des points.

En jouant sur $N$, $p$ et $s$, essayer d'améliorer la représentation, et tester différentes valeurs de $c$.

Le module `numba`, non disponible au lycée, permet de réduire le temps de calcul en optimisant la compilation. On placerait en début de code les lignes suivantes :

```
from numba import jit
@jit(nopython=True)
```

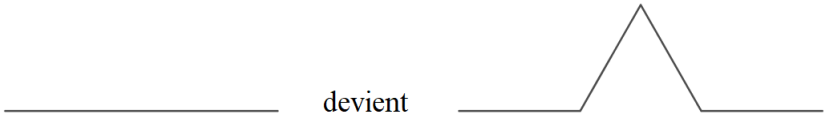
c. L'*ensemble de Mandelbrot* est étroitement lié aux ensembles de Julia : avec les notations précédentes, c'est l'ensemble des nombres complexes $c$ pour lesquels la suite $(z_n)$ est bornée lorsque $z_0 = 0$. Les ensembles de Julia les plus intéressants sont ceux pour lesquels $c$ est sur le bord de l'ensemble de Mandelbrot !

Écrire une fonction `mandelbrot(x,y)` qui renvoie **True** si la suite $(z_n)$ associée à $c = x + iy$ est bornée, **False** sinon. Procéder ensuite comme précédemment pour afficher l'ensemble de Mandelbrot.

## III. Le flocon de Koch

▷ **Exercice 4.**

Le *flocon de Koch* est une image fractale construite par récurrence. On applique à chaque côté d'un triangle équilatéral la transformation suivante :




puis on applique la même transformation à la figure obtenue, et ainsi de suite.

Pour tracer cette figure en Python, on va utiliser le module `turtle`. Ce module permet de simuler le déplacement de la pointe d'un stylo sur le plan. Celle-ci est initialement située à l'origine (0,0). Pour tracer un segment, on utilise les fonctions suivantes, à tester successivement dans le Shell :

```
>>> import turtle as t
>>> t.reset()
>>> print(t.pos())
>>> t.goto(100,100)
>>> print(t.pos())
>>> t.goto(300,-100)
>>> t.goto(400,0)
>>> t.exitonclick()
```

L'image obtenue au bout de $n$ transformations est donnée par la fonction :

```
import turtle as t
import math as m

def flocon(n):
    try:
        t.reset()    #réinitialisation de la figure
    except t.Terminator:    #permet d'éviter une exception bloquante
        pass
    cote(200,m.pi/3,n)     #tracé des cotés du triangle
    cote(200,-m.pi/3,n)
    cote(200,-m.pi,n)
    t.exitonclick()
```

où la fonction `cote(l,a,n)`, qui est utilisée pour tracer successivement les trois côtés de la figure, est une fonction récursive :

- si $n = 0$, on trace un simple segment de longueur $l$ et d'angle $a$ (par rapport à $[Ox)$)
- si $n > 0$, on fait appel quatre fois à la fonction `cote` sous la forme `cote(l_k,a_k,n-1)` où, pour tout $k \in [\![1,4]\!]$, $l_k$ et $a_k$ sont les longueurs et angles respectifs des segments obtenus par transformation du segment de longueur $l$ et d'angle $a$.

a. Écrire la fonction `cote`.
b. Tester la fonction `flocon` pour différentes valeurs de $n$.

# IV. La fractale de Newton

Soit $f$ une fonction dérivable sur un intervalle $[a,b]$ s'annulant exactement une fois en $\alpha$. On rappelle que la *méthode de Newton* consiste à déterminer numériquement $\alpha$ de la façon suivante : On choisit $u_0 \in [a,b]$ et on pose :

$$\forall n \in \mathbb{N} \qquad u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}.$$

La suite $(u_n)$, si elle converge, converge vers $\alpha$, et ceci de façon très rapide.

▷ **Exercice 5 : La fractale de Newton.**

On considère la fonction $f : x \mapsto x^3 - 1$, puis la suite associée à $f$ par la méthode de Newton avec $u_0 \in \mathbb{C}$. On admet que cette suite converge.

Sa limite est une des solutions de l'équation $f(x) = 0$, soit 1, $j = e^{i\frac{2\pi}{3}}$ ou $j^2 = e^{i\frac{4\pi}{3}}$.

Ceci définit trois zones du plan complexe $C(1)$, $C(j)$ et $C(j^2)$, un complexe $z$ étant dans la zone $C(\zeta)$ si la suite de premier terme $u_0 = z$ converge vers $\zeta$.

Ces trois zones sont appelées *bassins d'attraction*. Leurs frontières forment une fractale.

a. Écrire une fonction `bassin(z)` qui à un complexe $z$ associe 0, 1 ou 2 selon que la limite de la suite définie par la méthode de Newton soit 1, $j$ ou $j^2$.

b. Tracer les bassins d'attraction. Pour ceci :
   - Définir quatre réels $a$, $b$, $c$, $d$ qui déterminent la zone affichée $[a,b] \times [c,d]$.
     Poser initialement $(a,b) = (c,d) = (-1{,}5; 1{,}5)$.
   - Définir le nombre de points $N = 200$.
     Il sera remplacé par $N = 1000$ une fois le programme terminé.
   - Créer à l'aide de l'instruction `linspace` du module `numpy` le tableau des abscisses $X$ ($N$ points équirépartis de $a$ à $b$) et le tableau des ordonnées $Y$ ($N$ points équirépartis de $c$ à $d$).
   - Créer une matrice nulle de taille $(N,N)$ du type `array` de `numpy`.
   - Affecter tous les coefficients de cette matrice, le coefficient $(i,j)$ recevant la valeur 0, 1 ou 2 selon la limite de la suite définie par $u_0 = z$ où $z$ est le complexe défini par **complex**`(X[i],Y[j])`.
   - Afficher cette matrice à l'aide de l'instruction `imshow` de `matplotlib.pyplot`.

   On pourra utiliser la méthode de Newton avec une petite précision, comme $\varepsilon = 2^{-2}$.

c. Afficher quelques agrandissements. Par exemple la fenêtre $[0;0{,}5] \times [0;0{,}5]$, puis la fenêtre $[0{,}3;0{,}4] \times [0{,}2;0{,}3]$, etc.

d. Obtenir d'autres figures avec d'autres racines $n^{\text{èmes}}$ de l'unité.

# Travaux Pratiques n°11
# Représentation des entiers et des flottants

## I. Représentation des entiers

### Écriture binaire

Dans tout système informatique, les données sont codées en binaire. Pour les entiers, le passage du décimal au binaire, ou à l'hexadécimal, et inversement, s'effectue à l'aide des instructions suivantes :

```
>>> int('11010',2)  # écriture décimale du nombre binaire 11010
>>> bin(42)  # écriture binaire du nombre décimal 42
>>> hex(2022)  # écriture hexadécimale du nombre décimal 2022
```

En Python, les entiers de taille fixe (sur $n$ bits) correspondent aux types `uint8`, `uint16`, `uint32`, `uint64` du module `numpy` (*unsigned integer*, pour $n = 8$, 16, 32, 64 respectivement). Tester les instructions suivantes :

```
>>> import numpy as np
>>> x=np.uint16(2**16-1)
>>> type(x)
>>> x=x+1
>>> type(x)    # Qu'observe-t-on ?
```

### Entiers relatifs

Les entiers relatifs sont codés en *complément à deux* : on considère que tous les entiers sont codés sur $n$ bits, ce qui permet d'écrire $2^n$ entiers en tout, par exemple les entiers de 0 à $2^n - 1$ ou de $-2^{n-1}$ à $2^{n-1} - 1$.
Il est alors plus simple, vis-à-vis des opérations élémentaires, de les coder comme suit :

- un entier $k \in [\![0,2^{n-1} - 1]\!]$ est codé par sa représentation binaire,
- un entier $k \in [\![-2^{n-1}, -1]\!]$ est codé par la représentation binaire de $2^n + k = 2^n - |k|$.

▷ **Exercice 1 : Complément à deux.**

a. Écrire une fonction `comp_deux` prenant en argument un entier $k \in [\![-2^{15}, 2^{15} - 1]\!]$ et renvoyant, sous forme de chaîne de caractères binaire, la représentation de $k$ en complément à deux sur 16 bits.

b. Tester les instructions suivantes :

```
>>> x=np.uint16(-22)
>>> bin(x)   # on retrouve le code du complément à deux
```

Contrairement aux entiers de type `uint`, la taille des entiers de type **int** est uniquement limitée par la mémoire disponible. Cette capacité n'est cependant pas infinie, comme l'illustre l'exercice suivant.

▷ **Exercice 2 : puissances itérées de Knuth.** Les *puissances itérées de Knuth* généralisent la notion de puissance d'un nombre par un autre : étant donnés deux entiers $a$ et $b$, de même que $a \times b = \underbrace{a + \cdots + a}_{b \text{ fois}}$ puis que $a^b = \underbrace{a \times \cdots \times a}_{b \text{ fois}}$ : on note $a \uparrow b = a^b$, puis $a \uparrow\uparrow b = \underbrace{a \uparrow \cdots \uparrow a}_{b \text{ fois}}$, et plus généralement : $a \uparrow^n b = \underbrace{a \uparrow^{n-1} \cdots \uparrow^{n-1} a}_{b \text{ fois}}$, c'est-à-dire :

$$\forall (a,b,n) \in \mathbb{N}^2 \times \mathbb{N}^*, \quad a \uparrow^n b = \begin{cases} a^b & \text{si } n = 1, \\ 1 & \text{si } b = 0, \\ a \uparrow^{n-1} (a \uparrow^n (b-1)) & \text{sinon.} \end{cases}$$

a. Écrire une fonction `knuth(a,b,n)` qui renvoie $a \uparrow^n b$.

b. Calculer $3 \uparrow^n 2$ pour $n$ allant de 0 à 4. Qu'observe-t-on ?

## II. Représentation des flottants

Tout nombre réel $x$ s'écrit de façon unique :

$$x = \pm a \times 2^n \qquad \text{avec} \quad a \in [1, 2[ \quad \text{et} \quad n \in \mathbb{Z}.$$

On dit que $\pm$ est le *signe*, $a$ la *mantisse*, et $n$ l'*exposant*.

```
>>> x=1/7
>>> x.hex()
    # renvoie la mantisse de x en hexadécimal et l'exposant en décimal
   après le p
```

À la différence des entiers, la taille des nombres réels en Python est limitée par leur codage comme *nombres en virgule flottante*.

▷ **Exercice 3 : Tailles extrémales des flottants.**

a. En testant directement dans le Shell, déterminer le plus grand entier `p` pour lequel `2.**p` existe en Python. Attention à bien écrire `2.` (**float**) et non `2` (**int**).

b. De même, déterminer le plus grand entier `q` pour lequel `2.**(-q)` existe en Python. On a en fait $2^{-q} = \varepsilon \cdot 2^{1-p}$, où $\varepsilon$ est l'*unité d'arrondi* du microprocesseur, qui dépend donc du matériel utilisé. Cette quantité peut être déterminée par l'algorithme suivant :

$\varepsilon = 1.$
tant que $(1. + 0.5 * \varepsilon) \neq 1.$ :
    $\varepsilon = 0.5 * \varepsilon$

**Limites de précision des flottants**

Il est d'autre part impossible de représenter *exactement* des nombres réels dont le développement binaire est infini, ce qui peut rapidement mener à des erreurs d'arrondi, même sur des calculs très simples :

```
>>> x=0.3-0.1-0.2
>>> print(x)
>>> x==0
    # pour pallier ce problème, il vaut mieux remplacer le test ci-
   dessus par un test du type :
>>> abs(x) < 1e-15
```

On rencontre également des phénomènes d'*absorption*, lorsqu'on effectue des opérations sur des réels ayant des ordres de grandeur très différents :

```
>>> 2**100 + 1. - 2**100
```

ainsi que des phénomènes de *compensation*, lorsqu'on effectue la différence de deux nombres de même ordre de grandeur :

```
>>> f=lambda x:(x+1)**2-x**2-2*x-1
>>> [f(1.*10**(3*k)) for k in range(5)]
```

▷ **Exercice 4.** Quel résultat attendrait-on du programme suivant ? Comparer avec la figure effectivement obtenue :

```
import matplotlib.pyplot as plt

x=np.linspace(-1,1,100)
y=np.linspace(-2,2,100)

plt.plot(x,x+y-x-y)
plt.show()
```

▷ **Exercice 5.** On rappelle la convergence :

$$\lim_{n\to+\infty}\left(1+\frac{1}{n}\right)^n = e.$$

a. Calculer et afficher $\delta = \left|\left(1+\frac{1}{n}\right)^n - e\right|$ pour $n = 10^k$, $k$ allant de 0 à 30. La constante $e$ est présente dans le module `numpy`.

b. Représenter la courbe de $\delta$ en fonction de $k$. Expliquer le phénomène observé.

▷ **Exercice 6 : Nombres pseudo-aléatoires.** L'ordinateur est par construction une machine *déterministe*. Pour générer des nombres aléatoires, on utilise des algorithmes produisant des suites de nombres dont les propriétés *approchent* celles des suites aléatoires. On parle alors de *nombres pseudo-aléatoires*. Une telle suite est générée par la fonction `random` du module éponyme, par récurrence à partir d'une *graine*, produite par la fonction `seed`.

a. Exécuter le code suivant :

```
import random as rd
rd.seed(1)
for k in range(10):
    print(rd.random())
```

b. Exécuter à nouveau `rd.seed(1)`, puis `rd.random()` plusieurs fois. Qu'observe-t-on ?

c. Recommencer en remplaçant `rd.seed(1)` par `rd.seed(2)`.

d. Remplacer `rd.seed(1)` par `rd.seed()`. Qu'observe-t-on ?

## III. Arithmétique multi-précision

L'*arithmétique multi-précision* s'intéresse aux opérations sur les grands nombres :

- Leur stockage sous forme de liste ou, comme dans les exercices précédents, sous forme de chaîne de caractères, permet de n'être limité que par la mémoire disponible. Comme on l'a dit, ce n'est pas un problème pour les entiers en Python.
- On cherche des algorithmes plus efficaces que les algorithmes utilisés par la machine.

▷ **Exercice 7 : Algorithme de multiplication rapide de Karatsuba.** On souhaite multiplier deux grands entiers de la forme $a \times 10^k + b$ et $c \times 10^k + d$. Le développement standard du produit s'écrit :

$$(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + (ad + bc) \times 10^k + bd,$$

dont le calcul nécessite quatre multiplications ($ac$, $ad$, $bc$, $bd$). Or, on peut vérifier l'identité :

$$(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + (ac + bd - (a-b)(c-d)) \times 10^k + bd,$$

où trois multiplications suffisent ($ac$, $bd$, $(a-b)(c-d)$).
En utilisant cette écriture de façon récursive pour chacune des trois multiplications, et ainsi de suite, on obtient un algorithme sensiblement plus rapide que l'algorithme standard. On prend en général $k = \frac{p}{2}$, où $p$ est la longueur du plus grand des deux entiers.

On a donc, en notant $m \underset{K}{\cdot} n$ le produit de Karatsuba de $m$ et $n$ (qui sont considérés selon les cas comme des entiers ou comme des chaînes décimales) :

$$m \underset{K}{\cdot} n = \begin{cases} m \times n \text{ (produit standard)} & \text{si } m < 10 \text{ ou } n < 10, \\ (a \underset{K}{\cdot} c) \times 10^{2k} + \Big((a \underset{K}{\cdot} c) + (b \underset{K}{\cdot} d) - (a-b) \underset{K}{\cdot} (c-d)\Big) \times 10^k + (b \underset{K}{\cdot} d) & \text{sinon,} \\ \quad \text{où } k = \max(\text{len } m, \text{ len n})/2, & \\ \quad\quad a = m[:k],\ b = m[k:],\ c = n[:k], \text{ et } d = n[k:]. & \end{cases}$$

a. Écrire une fonction `kara(m,n)` qui reçoit deux entiers $m$ et $n$ et qui renvoie leur produit, en utilisant l'algorithme de Karatsuba.

b. En utilisant la fonction `time` du module éponyme, comparer les performances des algorithmes de multiplication usuel et de Karatsuba dans le calcul de $(10^{10^5} - 1)^2$.

# Travaux Pratiques n°12
# Complexité

La *complexité* d'un algorithme est l'ordre de grandeur du nombre d'opérations élémentaires de cet algorithme (calculs, affectations de variables, tests...) en fonction de la taille des arguments (grandeur d'un nombre, longueur d'une liste...)

Elle est représentative du *temps de calcul* de l'algorithme, ce qui est un enjeu crucial en informatique : si la taille des données est multipliée par dix, un algorithme de complexité linéaire mettra dix fois plus de temps à les traiter, mais un algorithme de complexité cubique en mettra mille fois plus !

Au cours de ce TP, nous allons estimer, expérimentalement et théoriquement, les complexités de différents algorithmes.

Le temps de calcul sera mesuré, en secondes, à l'aide de la fonction `process_time` de la librairie `time` :

```
t1=time.process_time()
fonction_a_evaluer(args)
t2=time.process_time()
print(t2-t1)
```

## I. Algorithmes sur des entiers

### ▷ Exercice 1 : Calcul de factorielle.

a. Écrire une fonction (de préférence itérative, et non récursive) `fact(n)` renvoyant $n!$.
b. Tester cette fonction sur de grands entiers, par exemple les $k * 1000$ pour $k \in [\![1,10]\!]$, en mesurant le temps de calcul.
c. Représenter le graphe du temps de calcul $t$ en fonction de $n$.
d. Retrouver théoriquement ce résultat en déterminant le nombre d'itérations de la fonction `fact` en fonction de $n$.

### ▷ Exercice 2 : Test de primalité.

a. Écrire une fonction `est_premier(n)` renvoyant **True** si $n$ est premier, **False** sinon.
b. Dresser une liste `P` de nombres premiers de la forme $a*10^b+c$ où $a \in [\![1,9]\!]$, $b \in [\![10,13]\!]$ et $c \in \{3,7,11,13,17,19\}$.
c. Tester la fonction `est_premier` sur les éléments `p` de `P`, en mesurant le temps de calcul.
d. Représenter le graphe du temps de calcul $t$ en fonction de $p$.

e. Retrouver ce résultat en déterminant le nombre d'itérations *maximal* de la fonction `est_premier` en fonction de $n$.
On parle alors de complexité *dans le pire des cas*. Quel est dans ce contexte le *meilleur des cas* ?

▷ **Exercice 3 : Exponentiation rapide.**

Étant donnés un réel $x$ et un entier naturel $n$, on veut écrire un algorithme renvoyant $x^n$ (sans utiliser l'opération `x**n` !). Dans un premier temps, on utilise la définition :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^{n-1} & \text{sinon} \end{cases} .$$

a. Écrire une fonction `exp(x,n)` renvoyant $x^n$. Déterminer son nombre d'itérations en fonction de $n$.

b. On fait à présent appel à l'*exponentiation rapide* : en utilisant le fait que :

$$x^n = \begin{cases} (x^{n//2})^2 & \text{si } n \text{ est pair} \\ x \times (x^{n//2})^2 & \text{si } n \text{ est impair} \end{cases} ,$$

écrire une fonction récursive `exp_rapide(x,n)` renvoyant $x^n$.

c. Déterminer le nombre d'itérations de `exp(x,n)` en fonction de $n$, ainsi que le nombre d'appels de `exp_rapide(x,n)` en fonction de $n$.

d. Tester ces deux fonctions sur de grandes valeurs de $n$, à $x$ fixé, par exemple : $x = 10^{200} - 1$ et $n = k * 10^2$ où $k \in [\![1,10]\!]$, en mesurant le temps de calcul.

e. Représenter sur un même graphe les temps de calcul $t$ et $t_r$ de ces deux fonctions en fonction de $n$. Pouvait-on s'attendre au résultat obtenu ?

▷ **Exercice 4 : La suite de Fibonacci.**

On rappelle la définition de la suite de Fibonacci $(F_n)$ :

$$\begin{cases} F_0 = 0,\ F_1 = 1 \\ \forall n \geq 0,\ F_{n+2} = F_{n+1} + F_n \end{cases} .$$

a. Écrire une fonction *itérative* `fibo_i(n)` codant cette définition et renvoyant $F_n$.

b. Écrire une fonction *récursive* `fibo_r(n)` codant cette définition et renvoyant $F_n$.

c. Tester ces deux fonctions pour $n \in [\![0,30]\!]$, en mesurant le temps de calcul.

d. Représenter sur un même graphe les temps de calcul $t_i$ et $t_r$ de ces deux fonctions en fonction de $n$. Qu'observe-t-on ?

e. En déterminant d'une part le nombre d'itérations de `fibo_i`, et d'autre part le nombre d'appels de `fibo_r`, en fonction de $n$, retrouver ce résultat.

## II. Algorithmes sur des listes

▷ **Exercice 5 : Plus grand élément d'une liste.**

a. Écrire une fonction **max**(`L`) renvoyant le plus grand élément d'une liste d'entiers `L`.

b. Créer des listes de grande longueur $n$, par exemple $n = k*10^6$ où $k \in [\![1,10]\!]$, contenant des entiers choisis aléatoirement dans $[\![1,1000]\!]$ (on utilisera la fonction `randint` de la librairie `random`).
Tester la fonction **max** sur ces listes, en mesurant le temps de calcul.

c. Représenter le graphe du temps de calcul $t$ en fonction de la longueur $n$ de la liste.

d. Retrouver théoriquement ce résultat en déterminant le nombre d'itérations de la fonction **max** en fonction de $n$.

▷ **Exercice 6 : Éléments communs à deux listes.**

a. Écrire une fonction `communs(A,B)` renvoyant les éléments communs à deux listes d'entiers `A` et `B`.

b. On suppose que $A$ et $B$ sont des listes de longueur $n$. Déterminer le nombre d'itérations de la fonction `communs` en fonction de $n$.

c. Tester la fonction `communs` sur des listes aléatoires de grande longueur $n$, par exemple $n = k * 10^3$ où $k \in [\![1,10]\!]$, en mesurant le temps de calcul.

d. Représenter le graphe du temps de calcul $t$ en fonction de $n$. Retrouve-t-on le résultat attendu ?

▷ **Exercice 7 : Recherche d'un élément dans une liste.**

a. Écrire une fonction `recherche_lineaire(c,L)` parcourant une liste *triée* (par ordre croissant) `L` et renvoyant `k` dès qu'elle trouve `L[k]=c`, **False** si `c` n'appartient pas à `L`.

b. Déterminer le nombre d'itérations de cette fonction, en fonction de la longueur $n$ de $L$, dans le pire des cas.

c. Écrire une fonction `recherche_dicho(c,L)` effectuant une recherche dichotomique de `c` dans `L` (toujours supposée triée) et renvoyant `k` si `L[k]=c`, **False** si `c` n'appartient pas à `L`.

d. Déterminer le nombre d'itérations de cette fonction, en fonction de la longueur $n$ de $L$, dans le pire des cas.

e. Tester ces deux fonctions sur des listes aléatoires de grande longueur, par exemple $n = k * 10^6$ où $k \in [\![1,10]\!]$, en mesurant le temps de calcul. Pour trier les listes, on utilisera la méthode `sort`.

f. Représenter sur un même graphe les temps de calcul $t_l$ et $t_d$ de ces deux fonctions en fonction de $n$. Retrouve-t-on le résultat attendu ?

# Travaux Pratiques n°13
# Tris

Nous avons rencontré dans les TP précédents l'algorithme de recherche dichotomique dans une liste. Cet algorithme, comme d'autres, ne peut opérer que sur une liste *ordonnée*, c'est-à-dire dont les éléments sont triés par ordre croissant.

Nous allons implémenter au cours de ce TP différents *algorithmes de tri* d'une liste non ordonnée, et comparer ces algorithmes entre eux.

## I. Création de listes-test et temps d'exécution

▷ **Exercice 1 : Générateur de listes.**

Afin de tester nos futurs algorithmes de tri, on commence par coder un générateur de listes de nombres. On utilise pour cela la fonction `randint` du module `random` :

```
from random import randint

for k in range(10):
    print(randint(1,6))
```

Écrire une fonction `gen(n,N)` renvoyant une liste aléatoire de $n$ entiers compris entre 0 et $N$.

▷ **Exercice 2 : Chronomètre.**

Pour tester l'efficacité de nos algorithmes, on chronomètre leur temps d'exécution à l'aide de la fonction `process_time` du module `time` :

```
from time import process_time

ping,pong,N=0,1,10**3

t1=process_time()
for k in range(N):
    ping,pong=pong,ping
t2=process_time()

print(t2-t1)
```

Tester le programme ci-dessus pour $N = 5,6,7,8...$ Qu'observe-t-on ?

Pour chaque algorithme présenté ci-dessous, la situation est la suivante : on veut trier une liste $L$ de $n$ nombres (ou de chaînes de caractères, ou de tout type pour lequel il existe une relation d'ordre).

## II. Tris en place

Les fonctions que l'on va écrire dans cette partie agissent directement sur la liste donnée en entrée pour la trier, sans stockage de liste auxiliaire. On parle de *tri en place*.

▷ **Exercice 3 : Tri par sélection.**

Le *tri par sélection* consiste à chercher le plus petit élément de la liste $L$ et à l'échanger avec l'élément en position initiale, puis à recommencer avec les éléments restants. Le pseudo-code est donc :

**Fonction tri par sélection**
  **Données :** $L$ (liste de longueur $n$)
  **Pour** $i$ de 0 à $n-2$ **faire**
    $j$ = indice du plus petit des $n-i$ derniers éléments de la liste
    $L[i] \leftrightarrow L[j]$
  **fin du Pour**
  **Résultat :** rien, mais la liste est ordonnée.
**fin de la fonction.**

a. Écrire une fonction `select(L)` permettant de trier la liste $L$ par sélection.
b. Tester et chronométrer cette fonction sur des listes aléatoires de taille 10, 100, 1000...
c. Dans quel cas y a-t-il le moins d'échanges à faire ? Dans quel cas y a-t-il le plus d'échanges à faire ? On les qualifie de *meilleur* et de *pire des cas*.

▷ **Exercice 4 : Tri par insertion.**

Le *tri par insertion* consiste à trier les éléments de la liste $L$ à mesure de la lecture de celle-ci : au moment de la lecture du $k^{\text{ème}}$ élément $L[k]$ de la liste, les $k-1$ premiers éléments de la liste sont déjà triés, et on insère alors $L[k]$ à sa place parmi les précédents :

**Fonction tri par insertion**
  **Données :** $L$ (liste de longueur $n$)
  **Pour** $k$ de 1 à $n-1$ **faire**
    Trouver $i \in [\![-1,k-1]\!]$ tel que $L[i] \leq L[k] \leq L[i+1]$ (avec $L[-1] = -\infty$)
    **Pour** $j$ de $i+1$ à $k-1$ **faire**
      $L[j+1] \leftarrow L[j]$ (attention à l'écrasement des variables)
    **fin du Pour**
    $L[i] \leftarrow L[k]$
  **fin du Pour**
  **Résultat :** rien, mais la liste est ordonnée.
**fin de la fonction.**

a. Écrire une fonction `insert(L)` permettant de trier la liste $L$ par insertion.
b. Tester et chronométrer cette fonction sur des listes aléatoires de taille 10, 100, 1000...
c. Dans quel cas y a-t-il le moins d'échanges à faire ? Dans quel cas y a-t-il le plus d'échanges à faire ?

▷ **Exercice 5 : Tri à bulles.**

Le *tri à bulles*, ou *tri par propagation*, consiste à comparer les éléments consécutifs de la liste, et à les échanger s'ils ne sont pas dans l'ordre.

**Fonction tri à bulles**
  **Données :** $L$ (liste de longueur $n$)
  **Pour** $i$ de $n-1$ à 1 **faire**
    **Pour** $j$ de 0 à $i-1$ **faire**
      **Si** $L[j+1] < L[j]$ **faire**
        $L[j] \leftrightarrow L[j+1]$
      **fin du Si**
    **fin du Pour**
  **fin du Pour**
  **Résultat :** rien, mais la liste est ordonnée.
**fin de la fonction.**

a. Écrire une fonction `bulles(L)` permettant de trier la liste $L$ par insertion.
b. Tester et chronométrer cette fonction sur des listes aléatoires de taille 10, 100, 1000...
c. Dans quel cas y a-t-il le moins d'échanges à faire ? Dans quel cas y a-t-il le plus d'échanges à faire ?
d. Estimer la complexité de cet algorithme en fonction de $n$.

## III. Tris récursifs

Les algorithmes de tris suivants sont récursifs. Ils ne sont pas *en place*, c'est-à-dire qu'ils utilisent une quantité importante de variables auxiliaires, et donc davantage de mémoire. En contrepartie, ils peuvent être plus rapides que les tris précédents.

▷ **Exercice 6 : Tri par partition-fusion.**

Le *tri par partition-fusion*, ou *tri fusion* est un algorithme dichotomique : si la liste $L$ contient un seul élément, elle est déjà triée ; sinon, on la sépare en deux sous-listes de même taille, que l'on trie par partition-fusion, puis on fusionne ces deux sous-listes triées afin d'obtenir une liste triée.

Cet algorithme utilise donc une fonction auxiliaire de fusion, elle-même récursive :

**Fonction fusion**
  **Données :** deux listes triées $A$ et $B$
  **Si** $A$ est vide **faire**
    Renvoyer $B$
  **sinon Si** $B$ est vide **faire**
    Renvoyer $A$
  **sinon Si** $A[0] < B[0]$ **faire**
    Renvoyer $[A[0]]$+fusion($A[1:]$, $B$)
  **sinonfaire**
    Renvoyer $[B[0]]$+fusion($A$, $B[1:]$)
  **fin du Si**
  **Résultat :** une liste triée, réunion de $A$ et $B$
**fin de la fonction.**

**Fonction tri fusion**
  **Données :** $L$ (liste de longueur $n$)
  **Si** $n \leq 1$ **faire**
    Renvoyer $L$
  **sinonfaire**
    Renvoyer fusion( tri fusion($L[:n//2]$), tri fusion($L[n//2:]$) )
  **fin du Si**
  **Résultat :** la liste $L$ triée
**fin de la fonction.**

a. Écrire la fonction `fusion(A,B)`, puis une fonction `tri_fusion(L)` permettant de trier la liste $L$ par partition-fusion.
b. Tester et chronométrer cette fonction sur des listes aléatoires de taille 10, 100, 1000...
c. Estimer la complexité de cet algorithme en fonction de $n$.

▷ **Exercice 7 : Tri rapide.**

Le *tri rapide* (ainsi baptisé par son inventeur, Charles Antony Richard Hoare), ou *tri pivot*, consiste à choisir un élément de la liste (le *pivot*) et à séparer les autres éléments de la liste en deux sous-listes selon qu'ils sont plus grands ou plus petits que le pivot. Ces deux sous-listes sont alors triées par tri rapide, puis réunies avec le pivot.

**Fonction tri rapide**
    **Données :** $L$ (liste de longueur $n$)
    **Si** $n \leq 1$ **faire**
        Renvoyer $L$
    **sinonfaire**
        pivot=$L[0]$
        $G$=éléments de $L \leq$ pivot
        $D$=éléments de $L >$ pivot
        Renvoyer tri rapide($G$) + [pivot] + tri rapide($D$)
    **fin du Si**
    **Résultat :** la liste $L$ triée
**fin de la fonction.**

a. Écrire une fonction `quick(L)` permettant de trier la liste $L$ par tri rapide.
b. Tester et chronométrer cette fonction sur des listes aléatoires de taille 10, 100, 1000...
c. Estimer la complexité de cet algorithme en fonction de $n$.
d. Que se passe-t-il lorsqu'on applique cet algorithme à une liste déjà triée ? Que faire dans ce cas ?

# IV. Autres algorithmes de tri

▷ **Exercice 8 : Tri par comptage.**

Le *tri par comptage*, ou *tri casier* est seulement utilisable sur des listes $L$ d'entiers, et plus particulièrement adapté lorsque $n > N$ (avec les notations du I.). Il consiste à créer une *table de comptage* $T$ de longueur $N+1$ telle que pour tout $k \in [\![0,N]\!]$, $T[k]$ soit égal au nombre d'occurrences de $k$ dans $L$. Il suffit pour cela de parcourir $L$, et il est ensuite immédiat de construire la liste triée de $L$ à partir de $T$.

**Fonction tri par comptage**
**Données :** $L$ (liste de longueur $n$ d'entiers de $[\![0,N]\!]$)
$T$=liste nulle de taille $(N+1)$
**Pour** $i$ de 0 à $n-1$ **faire**
Incrémenter $T[L[i]]$
**fin du Pour**
Reconstituer la liste $L$ triée à partir de $T$
**Résultat :** Liste ordonnée contenant $T[k]$ fois $k$ pour $k$ de 0 à $N$.
**fin de la fonction.**

a. Écrire une fonction `comptage(L)` permettant de trier la liste $L$ par comptage.
b. Tester et chronométrer cette fonction sur des listes aléatoires de taille 10, 100, 1000...

▷ **Exercice 9 : Tri par paquets.**

Le *tri par paquets*, quant à lui, s'applique à des listes $L$ de nombres réels compris dans un segment $[a,b]$ donné. On va supposer qu'il s'agit de [0,1]. L'idée est de créer $p$ paquets $L_0,\dots,L_{p-1}$, puis de répartir les éléments de $L$ entre les paquets, le paquet $L_k$ recevant les nombres appartenant à l'intervalle $\left[\frac{k}{p},\frac{k+1}{p}\right[$. On trie alors chaque paquet (avec l'un des cinq algorithmes vus ci-dessus), puis on réunit les paquets.

**Fonction tri par paquets**
**Données :** $L$ (liste de longueur $n$ de réels de [0,1])
$L_0,\dots,L_{p-1}$=listes vides
Répartir les éléments de $L$ entre les paquets
Trier les paquets
Concaténer les paquets
**Résultat :** la liste $L$ triée
**fin de la fonction.**

a. Adapter la fonction `gen` au cas présent, en utilisant la fonction `random` du module `random`.
b. Écrire une fonction `bucket(L)` permettant de trier la liste $L$ par paquets.
c. Tester et chronométrer cette fonction sur des listes aléatoires de taille 10, 100, 1000...
d. Expliquer, dans le cas général où les nombres sont compris dans un segment $[a,b]$, comment se ramener au cas de [0,1].

# Travaux Pratiques n°14
# Graphes

## I. Implémentation des graphes

▷ **Exercice 1 : Représentation des graphes.** Rappelons qu'un graphe orienté est constitué de *sommets* et d'*arcs* reliant ces sommets. Dans tout le TP, on représentera l'ensemble des sommets d'un graphe par la liste des entiers de 0 à $n-1$ (où $n$ est le nombre de sommets du graphe). L'ensemble des arcs sera représenté par une liste de couples d'entiers $(i,j)$ où $i$ et $j$ sont des sommets du graphe.

a. Tracer à la main les graphes orientés définis par les listes suivantes :

```
sommets=[k for k in range(4)]
arcs1=[(0,1),(1,2),(2,3),(3,0)]
arcs2=[(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)]
G1=[sommets,arcs1]
G2=[sommets,arcs2]
```

b. Il est possible de représenter les graphes en Python à l'aide du module `networkx`. Implémenter les lignes suivantes (si le module `networkx` n'est pas installé, passer à la question suivante) :

```
import networkx as nx
# si le module n'est pas installé, on peut utiliser la commande
# >>> pip install networkx
# (non disponible sur les ordinateurs du lycée)
import matplotlib.pyplot as plt

RG1=nx.DiGraph()   # pour les graphes non-orientés : RG=nx.Graph()
RG1.add_nodes_from(sommets)
RG1.add_edges_from(arcs1)

nx.draw(RG1,with_labels=True,node_color='cyan')
plt.show()
```

c. Écrire une fonction `matrice_adjacence(G)` qui prend en argument un graphe et renvoie sa *matrice d'adjacence*, c'est-à-dire $M \in M_n(\mathbb{R})$ telle que $m_{ij} = 1$ si $(i,j)$ est un arc du graphe, 0 sinon. On pourra utiliser la fonction `zeros` du module `numpy`. Tester cette fonction sur les graphes précédents.

d. Reprendre les questions précédentes dans le cas des graphes non-orientés (les arcs sont alors appelés des *arêtes*). Écrire une fonction `voisins(G,k)` qui prend en argument un entier $k$ et renvoie la liste des sommets $i$ voisins du sommet $k$ dans le graphe $G$ (c'est-à-dire tels que $(i,k)$ ou $(k,i)$ appartienne à l'ensemble des arêtes de $G$).

## II. Parcours de graphes

On considère dans la suite du TP des graphes non-orientés. Pour tester les fonctions, on pourra utiliser le fichier `villes.py` disponible sur `prepabellevue.org`.

Le *parcours d'un graphe* consiste à visiter (une et une seule fois) chaque sommet du graphe afin de résoudre un problème donné. Contrairement au parcours de liste dont nous avons l'habitude, qui est naturellement linéaire, le parcours de graphe impose, de par la structure du graphe, de faire des choix dans l'ordre de visite des sommets.

▷ **Exercice 2 : Parcours en profondeur.** Une première possibilité consiste à parcourir le graphe *en profondeur*, c'est-à-dire à :

- *visiter* le sommet où l'on se trouve, c'est-à-dire l'ajouter à une liste `visites` et *marquer* ses voisins non-visités, c'est-à-dire les ajouter en tête d'une liste `marques`,
- réitérer l'opération sur les sommets marqués jusqu'à ce qu'il n'y en ait plus.

Marquer les sommets consiste ici à les ajouter à une *pile*, c'est-à-dire en *première position* de la liste `marques`. Le sommet en première position de cette liste est alors le prochain sommet à visiter. Le parcours se termine lorsque tous les voisins ont été visités.

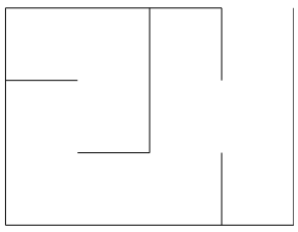
a. Écrire une fonction `voisins_restants(G,s,visites)` qui renvoie la liste des voisins du sommet $s$ qui n'ont pas encore été visités.

b. Écrire une fonction `parcours_en_profondeur(G,s)` qui prend en argument un graphe $G$ et un sommet de départ $s$, et renvoie la liste des sommets du graphe parcouru en profondeur à partir de $s$.

c. Application : Sortir d'un labyrinthe
On modélise un labyrinthe de taille $m \times n$ à l'aide d'un graphe de la façon suivante :
   - Les cases du labyrinthe sont numérotées de 0 à $mn - 1$ (de gauche à droite puis de haut en bas) et constituent les sommets du graphe,
   - Si on peut passer d'une case à une autre (c'est-à-dire si elles sont voisines et ne sont pas séparées par un mur), on met une arête entre les sommets correspondants.

   On suppose que la sortie du labyrinthe est la case en haut à droite, c'est-à-dire de numéro $n - 1$.

   i. Implémenter le graphe correspondant au labyrinthe suivant :



   ii. Écrire une fonction `sortir(G,k)` qui prend en argument la case $k$ du labyrinthe et qui renvoie une liste `chemin` de sommets permettant de sortir du labyrinthe à partir de la case $k$. On utilisera le parcours en profondeur, en prenant soin, dans la liste `chemin`, d'enlever les pistes menant à une impasse lorsqu'on rebrousse chemin (c'est-à-dire lorsqu'on dépile).

▷ **Exercice 3 : Parcours en largeur.** Une autre possibilité consiste à parcourir le graphe *en largeur*. Marquer les sommets consiste ici à les ajouter à une *file*, c'est-à-dire en *dernière* position de la liste `marques`. Comme précédemment, le sommet en première position de cette liste est le prochain sommet à visiter. Le parcours se termine lorsque la file est vide.

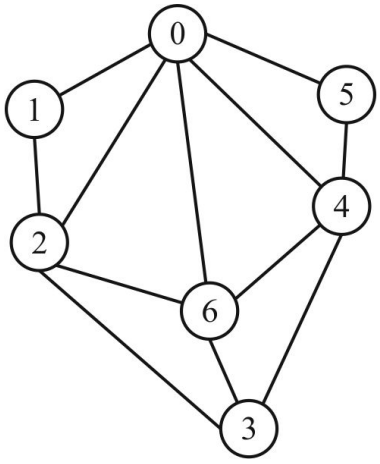
Écrire une fonction `parcours_en_largeur(G,s)` qui prend en argument un graphe $G$ et un sommet de départ $s$, et renvoie la liste des sommets du graphe parcouru en largeur à partir de $s$.

▷ **Exercice 4 : Coloration d'un graphe.** La *coloration* d'un graphe consiste à colorier les sommets d'un graphe, de sorte que deux sommets voisins n'aient jamais la même couleur, en utilisant au total un minimum de couleurs.

Pour obtenir une coloration proche de l'optimale, on utilise l'*algorithme de Welsh-Powell* :

- On classe les sommets dans l'ordre décroissant de leur degré (c'est-à-dire du nombre de leurs voisins)
- On parcourt cette liste, dans l'ordre, plusieurs fois. À chaque parcours, on choisit une nouvelle couleur, qu'on attribue au premier sommet non-coloré ainsi qu'à tous les autres sommets non-colorés qui ne sont pas voisins d'un sommet de cette couleur.
- On s'arrête lorsque tous les sommets sont colorés.

a. Appliquer cet algorithme, à la main, au graphe suivant :




b. Écrire une fonction `degre(G,k)` qui prend en argument un graphe $G$ et un sommet $k$ et renvoie le degré du sommet $k$.

c. Écrire une fonction `trier(G)` qui renvoie la liste des sommets du graphe, triés dans l'ordre des degrés décroissants.

d. Écrire une fonction `colorer(G)` qui met en œuvre l'algorithme de Welsh-Powell, et renvoie la liste des couples `(sommet,couleur)` (on représentera les couleurs par des entiers naturels).

## III. Recherche de plus courts chemins

Un problème important en théorie des graphes est celui du *plus court chemin*. Il s'agit en pratique de déterminer le trajet optimal entre deux villes, d'évaluer la proximité entre deux membres d'un réseau social, etc. Dans cette partie, les graphes sont supposés non-orientés et pondérés.

▷ **Exercice 5 : Algorithme de Dijkstra.** L'*algorithme de Dijkstra* détermine l'ensemble des plus courtes distances à un sommet donné $s_0$. Il consiste à parcourir le graphe à partir de $s_0$, en mettant progressivement à jour les distances des sommets rencontrés.

a. Initialement, seul le sommet $s_0$ est connu et tous les sommets (sauf $s_0$) sont supposés à une distance infinie de $s_0$. Écrire une fonction `init_dijkstra(G,s0)` renvoyant le couple $(H,D)$, où $H$ est le graphe réduit au sommet $s_0$ et $D$ la liste des distances initialement connues (on utilisera pour l'infini la constante `inf` du module `numpy`).

b. Écrire une fonction `suivant_dijkstra(G,H)` prenant en argument le graphe $G$ et un sous-graphe $H$ de $G$, et renvoyant le triplet `(p,s,d)` où $s$ est le sommet de distance minimale $d$ à $H$, et $p$ est un sommet de $H$ tel que l'arête $(p,s)$ est de poids $d$ ; ou `(-1,-1,inf)` si $H = G$.

c. Écrire une fonction `maj_dijkstra(G,H,D,p,s,d)`, qui au sous-graphe $H$ et à la liste $D$ :

- rajoute à $H$ le sommet $s$ et les arêtes reliant ce sommet aux autres sommets du sous-graphe,
- - pose $D[s] = D[p] + d$,
  - remplace, pour tout sommet $k$ voisin de $s$, $D[k]$ par $D[s] + w$ où $w$ est le poids de l'arête $(s,k)$, si le chemin passant par $s$ est plus court,
  - remplace de même $D[s]$ par $D[k] + w$ où $w$ est le poids de l'arête $(s,k)$, si le chemin passant par $k$ est plus court.

d. Écrire une fonction `dijkstra(G,s0)` renvoyant la liste $L$ des plus courtes distances à $s_0$. On parcourra pour cela le graphe $G$ à partir de $s_0$ dans l'ordre donné par la fonction `suivant_dijkstra`, en mettant à jour à chaque étape le sous-graphe $H$ et la liste $D$ grâce à la fonction `maj_dijkstra`.

e. Tester la fonction sur un trajet Toulouse-Strasbourg.

▷ **Exercice 6 : Algorithme de Floyd-Warshall.** Si on s'intéresse plus généralement à l'*ensemble* des plus courtes distances entre deux sommets quelconques du graphe, l'*algorithme de Floyd-Warshall* est tout indiqué. Il repose sur le principe suivant : étant donnés trois sommets $i$, $j$, $k$ du graphe, le plus court chemin (s'il existe), de longueur $d_{ij}^{(k)}$, allant de $i$ à $j$ et n'utilisant que les sommets de 1 à $k$ :

- ne passe pas par $k$ et donc n'utilise que les sommets de 1 à $k-1$ : $d_{ij}^{(k)} = d_{ij}^{(k-1)}$,
- **ou** passe par $k$, et est alors la concaténation des plus courts chemins connus allant de $i$ à $k$ et de $k$ à $j$ : $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

En d'autres termes, la matrice des plus courts chemins peut être obtenue par :

**Algorithme de Floyd-Warshall**

**Données :** $G$ (graphe pondéré non-orienté à $n$ sommets)
$D \leftarrow$ Matrice d'adjacence de $G$
**Pour** $k$ de 0 à $n-1$ **faire**
  **Pour** $i$ de 0 à $n-1$ **faire**
    **Pour** $j$ de 0 à $n-1$ **faire**
      $d_{ij} \leftarrow \min(d_{ij}, d_{ik} + d_{kj})$
**Résultat :** $D$ (matrice des plus courtes distances de $G$).

a. Réécrire la fonction `matrice_adjacence(G)` afin que $m_{ij}$ soit égal au poids de l'arête $(i,j)$ si elle existe, `inf` sinon.
b. Écrire une fonction `floyd_warshall(G)` mettant en œuvre cet algorithme sur un graphe $G$.
c. Tester la fonction sur les graphes précédents.
d. Estimer la complexité de cet algorithme, et la comparer avec la complexité de l'algorithme de Dijkstra.

### Représentation des graphes pondérés

```
sommets=[k for k in range(4)]
arcs3=[(1,2,10),(0,2,5),(2,3,7)]
G3=[sommets,arcs3]

RG3=nx.DiGraph()
RG3.add_nodes_from(sommets)
RG3.add_weighted_edges_from(arcs3)

pos=nx.spring_layout(RG3)
w = nx.get_edge_attributes(RG3, "weight")
nx.draw_networkx(RG3,pos,with_labels=True,node_color='cyan')
nx.draw_networkx_edge_labels(RG3,pos,edge_labels=w)
plt.show()
```