Révisions - Éléments de correction

- N'hésitez pas à copier-coller pour tester rapidement : fonctionne beaucoup mieux avec la version `html`
- Si vous utilisez Pyzo, pensez à utiliser les cellules (`## ...`) et `Ctrl+Entrée`
- Essayez par vous-même ; tentez des variantes ; essayez sur des exemples concrets...

1 Chercher un élément dans une liste

1.1 Linéaire

1.2 Dichotomie

- Si la liste en entrée est triée, on peut utiliser l'algoritme de dichotomie
- Bien faire attention : `d`, `f` et `m` sont des indices (numéros de cases) et pas des valeurs de la liste ! On compare `x` à `t[m]` par exemple

```
def dicho(x, t):
    d, f = 0, len(t)-1   # Convention : `d` et `f` inclus
    while d <= f:
        m = (d+f)//2     # Milieu : formule à connaître !!! Ici, division entière.
        if x == t[m]:
            return m     # x trouvé en position 'm' ; on peut aussi renvoyer True
        elif x < t[m]:
            f = m-1
        else:
            d = m+1
    return None          # pas trouvé ; on peut renvoyer False


t = [-3, 5, 7, 12, 99]
print(dicho(2, t))
print(dicho(12, t))

# ça existe dans Python
# module `bisect` un peu compliqué à utiliser ; autant savoir le réécrire
```

Complexité :

- taille de la zone considérée (au départ : `f-d+1` $= n$, la taille de la liste `t`).
- Après k étapes : plus petites que $\frac{n}{2^k}$
- On cherche la plus petites valeur de k telle que : $\frac{n}{2^k} \leq 1$
- On obtient $k \geq \frac{\ln(n)}{\ln(2)}$, noté $k = \log_2(n)$

2 Recherche du maximum

2.1 Valeur du maximum

2.2 Position du maximum

Si on veut la position, il *faut* utiliser une boucle `for` avec `range(len(t))` ou une boucle `while` sur les indices :

```
def pos_max(t):
    """ Renvoie la position du maximum de la liste t """
    pos = 0
    for i in range(len(t)):      # Syntaxe à connaître
        if t[i] > t[pos]:
```

```
            pos = i
    return pos

print(pos_max([-9, 12, 4, -1, 2]))
```

2.3 Variantes

Dans les sujets de concours, on peut vous demander des variantes, comme :

- Pour une liste de couples, renvoyer les couples qui ont la plus grande deuxième composante (il peut y avoir égalité)

```
def pos_max_couples(t):
    res = []
    maxi = t[0][1]            # deuxième composante du premier élément
    for i in range(len(t)):
        if t[i][1] > maxi:    # Nouveau plus grand élément : on repart à zéro
            res = [t[i]]
            maxi = t[i][1]
        elif t[i][1] == maxi: # on a trouvé un ex-aequo
            res.append(t[i])
    return res

print(pos_max_couples([('A', 1), ('B', 5), ('C', -2), ('D', 5), ('E', 3)]))
```

3 Moyenne, Variance, ...

C'est l'occasion d'apprendre à réutiliser des fonctions

```
def somme(t):
    s = 0
    for x in t:
        s += x
    return s

def moyenne(t):
    return somme(t)/len(t)

def variance(t):              # Complexité : O(n), même en prenant en compte les appels de fonctions
    moy = moyenne(t)
    carres_ecarts = [(x-moy)**2 for x in t]
    return moyenne(carres_ecarts)

u = [-9, 12, 4, -1, 2]
print(moyenne(u))
print(variance(u))
```

On a utilisé la formule de la "moyenne des carrés des écarts à la moyenne) : https://fr.wikipedia.org/wiki/Variance_(math%C3%A9matiques)

4 Intégration

On calcule des sommes de Riemann (ici **à gauche**) pour la fonction f sur un intervalle $[a;b]$ avec n subdivisions (toutes de taille $h = \frac{b-a}{n}$). Pour simplifier, on notera $\sigma_k = a + k \times \frac{b-a}{n}$.

$$R_n = \frac{b-a}{n} \sum_{k=0}^{n-1} f(a + k \times \frac{b-a}{n})$$

```
def rect(f, a, b, n):
    """ Intégration : rectangles. Somme de Riemann à gauche. """
    s = 0
    h = (b-a)/n
    for k in range(n):    # n exclu
```

```
        sigma = a + k * h
        s += f(sigma)
    return h * s

def g(x):
    return x**2

print(rect(g, 0, 1, 100))

# ça existe dans Python
from scipy.integrate import quad
print(quad(g, 0, 1))              # renvoie un couple : la valeur calculée et une estimation de l'erreur
```

5 Résolution d'équation

On se ramène toujours à la résolution d'équation du type $f(x) = 0$, même dans les cas compliqués.

5.1 Dichotomie

- Ça s'appelle aussi dichotomie ("couper en deux"), mais le fonctionement est un peu différent de la dichotomie sur les listes : la fonction n'a pas besoin d'être monotone sur l'intervalle.
- Il faut quand même vérifier avant les conditions d'application du Théorème des Valeurs Intermédiaires.
- On utilise l'astuce : $f(x) \times f(y) \leq 0$ ssi $f(x)$ et $f(y)$ n'ont pas le même signe

```
def solve_dicho(f, a, b, epsilon):
    """ Solution de f(x)=0 sur l'intervalle [a; b] à epsilon près. """
    while b - a > epsilon:
        milieu = (a+b)/2          # Division sur les flottants. Important !
        if f(a)*f(milieu) <= 0:   # Si f(a) et f(milieu) n'ont pas le même signe, on continue à gauche
            b = milieu
        else:
            a = milieu
    return (a+b)/2

def h(x):
    return x**2-2

print(solve_dicho(h, 1, 2, 1e-15))

# ça existe dans Python
from scipy.optimize import bisect
print(bisect(h, 1, 2))
```

5.2 Méthode de Newton

- Avantage de la dichotomie : on est sûr d'obtenir une solution (si elle existe) sur l'intervalle $[a; b]$
- Avantage de la méthode de Newton :
  - quand elle fonctionne (cf. cours de Maths), elle est très très rapide.
  - On peut l'utiliser sur des fonctions qui manipulent de nombreux objets mathématiques : Complexes, Matrices, ... !
- Inconvénients de la méthode de Newton : a priori, il faut connaître la dérivée f' de la fonction considérée. Peut ne pas converger, ou ne pas fonctionner du tout par exemple si la dérivée s'annule.

On part d'un x_0 fixé par l'utilisateur, puis on calcule les termes de la suite

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

```
def newton(f, x, fprime, epsilon):
    while abs(f(x)/fprime(x)) > epsilon:
        x = x - f(x)/fprime(x)
    return x

def h(x):
```

```
    return x**2-2

def hprime(x):
    return 2*x

print(newton(h, 5, hprime, 1e-15))  # x_0 = 5 a été choisi de manière totalement arbitraire

# ça existe dans Python
from scipy.optimize import fsolve
print(fsolve(h, 5))                 # renvoie une liste à un élément, c'est comme ça
```

Ici on a utilisé la condition d'arrêt : $|x_{k+1} - x_k| \leq \varepsilon$ qui n'est pas toujours le choix le plus judicieux, mais est très utilisé.

6 Équations différentielles

6.1 Méthode d'Euler

- On ne considère que des équations différentielles qu'on peut mettre sous la forme $y'(t) = F(y(t), t)$ où F signifie ici "est une fonction de ..."
- Pour toute fonction y de classe $\mathcal{C}^1$ sur un intervalle $[t_k; t_{k+1}]$, on a $y(t_{k+1}) = y(t_k) + (t_{k+1} - t_k) \times \int_{t_k}^{t_{k+1}} F(y(t), t)dt$
- **Approximation** (Méthode d'Euler explicite) : sur tout l'intervalle on assimile $F(y(t), t)$ à $F(y(t_k), t_k)$ et on obtient

$$y_{k+1} = y_k + (t_{k+1} - t_k) \times F(y_k, t_k)$$

- à partir de maintenant, on considère vraiment des points dans le plan, d'abscisse t_i et d'ordonnée y_i.

```
import numpy as np
import matplotlib.pyplot as plt

def euler(F, y0, temps):
    y = np.array(y0)
    res = [y]                       # 'res' est une liste python
    for k in range(len(temps)-1):
        y = y + (temps[k+1]-temps[k]) * np.array(F(y, temps[k]))
        res.append(y)
    return np.array(res)            # type renvoyé : 'np.ndarray'

# Données spécifiques à l'équation différentielle considérée

def F(y, t):
    """ EquaDiff : y'(t) = 2*y(t)+sin(t) """
    return 2*y+np.sin(t)   # Remarque : ici on utilise `y` et PAS `y(t)`

initial = 1       # y_0

# Résolution et Affichage

t = np.linspace(0, 1, 100)  # 100 points équirépartis sur [0; 1]
y_euler = euler(F, initial, t)
plt.plot(t, y_euler, '+', label="Méthode d'Euler")

# ça existe dans Python
from scipy.integrate import odeint
y_odeint = odeint(F, initial, t)
plt.plot(t, y_odeint, label="odeint")


plt.legend()
plt.show()
```

6.2 Vectorialisation

- Remarque : le code ci-dessus fonctionne aussi avec la vectorialisation

- C'est avant tout une histoire de notations. Si on a l'EquaDiff : $\frac{d^2\theta}{dt^2} = -\frac{g}{L}\theta$:
- On pose $Y = \begin{pmatrix}\theta\\ \theta'\end{pmatrix}$ et on veut écrire l'EquaDiff sous la forme $Y' = F(Y,t)$. On obtient

$$\begin{pmatrix}\theta'\\ \theta''\end{pmatrix} = F(\begin{pmatrix}\theta\\ \theta'\end{pmatrix}, t) = \begin{pmatrix}\theta'\\ -\frac{g}{L}\theta\end{pmatrix}$$

```
# variables globales
g = 9.81    # m.s^-2
L = 1       # m

def F(Y, t):
    y, yp = Y     # décompose le couple 'Y' en deux composantes nommées 'y' et 'yp' (unpacking)
    return (yp, -g/L*y)

initial = (1, 1)      # (y_0, y'_0)
```

et on peut utiliser directement les fonctions ci-dessus. On obtient deux courbes : une pour θ et une pour θ'

6.3 Variantes

Il en existe de nombreuses :

- Méthode d'Euler implicite
- Méthode de Heun, Runge-Kutta
- Méthode de Verlet

7 Gauss

- La méthode présentée ici est la plus simple : on se ramène à la matrice identité. Il existe une variante (à connaître) qui utilise une matrice triangulaire supérieure, puis "remonte" en résolvant des équations.
- On a besoin de quelques outils. Évidement tout est faisable directement en utilisant `numpy`
- On utilise la "recherche partielle de pivot" pour obtenir le numéro de la ligne qui nous intéresse pour le pivot : c'est une recherche de position de maximum classique, mais un peu subtile.

```
def nulle(n, p):
    return [[0]*p for i in range(n)]  # Important, à connaître

def dim(A):
    return len(A), len(A[0])

def copie(A):
    n, p = dim(A)
    res = nulle(n, p)
    for i in range(n):
        for j in range(p):
            res[i][j] = A[i][j]
    return res

def pivot_partiel(A, j):
    """ Dans la colonne 'j', renvoie la ligne 'i' qui correspond à
    la plus grande valeur (en valeur absolue) sous la diagonale (au sens large) """
    i = j
    for k in range(j, len(A)):
        if abs(A[k][j]) > abs(A[i][j]):
            i = k
    return i

def echanger(A, i1, i2):
    for j in range(len(A[0])):
        A[i1][j], A[i2][j] = A[i2][j], A[i1][j]

def transvection(A, i, j, mu):
```

```
    """ Li <- Li + mu * Lj """
    for k in range(len(A[0])):
        A[i][k] = A[i][k] + mu * A[j][k]

def dilatation(A, i, coeff):
    for k in range(len(A[0])):
        A[i][k] = coeff * A[i][k]

def gauss(A, B):
    A = copie(A)
    n, p = dim(A)
    for j in range(p):  # pour chaque colonne
        ligne_pivot_partiel = pivot_partiel(A, j)
        echanger(A, j, ligne_pivot_partiel)
        echanger(B, j, ligne_pivot_partiel)
        for i in range(n):
            if i != j:   # pas sur la diagonale
                mu = -A[i][j]/A[j][j]
                transvection(A, i, j, mu)
                transvection(B, i, j, mu)
            else:
                coeff = 1/A[i][i]
                dilatation(A, i, coeff)
                dilatation(B, i, coeff)
    return A, B  # a priori A est égale à l'identité

A = [[1, 2], [3, 4]]
B = [[1, 0], [0, 1]]
print(gauss(A, B))

# ça existe en Python
from numpy.linalg import inv
print(inv(A))
```

8 Chaînes de caractères

C'est important de savoir manipuler les chaines de caractères en Python

- On peut les lire avec la même syntaxe que les listes : `s[0]`, `s[-1]`, `s[2:5]`, `for caractere in s:`...
- MAIS on ne peut pas les modifier (non mutable), donc on utilisera souvent `s = s + "..."` même si ce n'est pas aussi efficace que `join`
- Pour chercher une chaine (`mot`) dans une autre (`chaine`), c'est plus simple d'utiliser une fonction auxiliaire :

```
def apparait_pos(mot: str, chaine: str, d: int) -> bool:
    """ Est-ce que 'mot' apparait dans 'chaine' à partir de la position 'd' """
    for i in range(len(mot)):
        if mot[i] != chaine[d+i]:
            return False
    return True

def cherche_mot(mot, chaine):
    for decalage in range(len(chaine)-len(mot)+1):
        if apparait_pos(mot, chaine, decalage):
            return decalage                          # ou True
    return -1                                        # ou False

mot, chaine = "jour", "Bonjour"
print(cherche_mot(mot, chaine))

# ça existe en python
print(mot in chaine)
print(chaine.find(mot))
```

9 Fonctions récursives

- Les super classiques : ne pas oublier les cas de base ! (cf. `TP-Recursivite.pdf`) (Exponentiation rapide, Factorielle, Fibonacci...)

10 Tris

- Les trois tris suivants sont au programme : en particulier le "tri par insertion" qui peut ne pas sembler important mais qui peut apparaître au concours sans que ce soir explicite
- Pour montrer qu'un algorithme de tri fonctionne il faut montrer :
  1. Que le résultat contient exactement les mêmes que l'entrée. Évident si on ne fait que des échanges.
  2. Que le résultat est trié. On utilisera souvent un invariant de boucle.
- Pour les complexités : cf. la suite

```
# ça existe dans Python
t = [3, -12, 9, 4]
u = sorted(t)     # Renvoie une nouvelle liste, triée. Ne modifie pas t
t.sort()          # Ne renvoie rien (`None`). Modifie `t`
```

10.1 Tri par insertion

```
def tri_insertion(t):
    """ Tri par insertion. En place. Complexité : O(n^2). """
    for i in range(1, len(t)):
        j = i
        while j > 0 and t[j-1] > tab[j]:
            t[j-1], t[j] = t[j], t[j-1]
        j -= 1
```

10.2 Tri rapide

```
from random import randint

def tri_rapide(t):
    if len(t) <= 1:
        return t
    pivot = t[randint(0, len(t) - 1)]   # Choix du pivot au hasard
    petits, egaux, grands = [], [], []
    for x in t:                         # Découpage en fonction du pivot
        if x < pivot:
            petits.append(x)
        elif x == pivot:
            egaux.append(x)
        else:
            grands.append(x)
    return tri_rapide(petits) + egaux + tri_rapide(grands)
```

10.3 Tri fusion

Le coeur du tri fusion est l'opération de `fusion` de deux listes qui sont déjà triées. On veut que le résultat aussi soit trié. Et pour garantir une bonne complexité du tri fusion, cette opération doit s'exécuter en $O(|u| + |v|)$

```
def fusion(u, v):
    res, i, j = [], 0, 0
    while i < len(u) and j < len(v):  # Ici on utilise 'and' : on sortira dès qu'on arrive à la fin d'une des de
        if u[i] < v[j]:
            res.append(u[i])
            i += 1
        else:
            res.append(v[j])
            j += 1
    while i < len(u):    # soit cette condition est fausse dès qu'on sort de la première boucle while
```

```
        res.append(u[i])
        i += 1
    while j < len(v):    # soit c'est celle-ci
        res.append(v[j])
        j += 1
    return res

def tri_fusion(t):
    if len(t) <= 1:
        return t
    m = len(t)//2
    return fusion(tri_fusion(t[:m]), tri_fusion(m:))
```

11 Pgcd

```
def pgcd_euclide(a, b):
    while b != 0:
        a, b = b, a%b
    return a
```

12 Nombres premiers

```
from math import sqrt

def premier(n: int) -> bool:
    if n == 0 or n == 1:
        return False
    for d in range(2, int(sqrt(n))+1):
        if n%d == 0:
            return False
    return True
```

13 Complexité : boucles

- Prendre en compte les boucles `for` (facile) et `while` (pas toujours évident)
- Prendre en compte les appels de fonctions ! (important)
- Si on veut détailler : pour une boucle `for i in range(a, b):`, utiliser $\sum_{i=a}^{b-1}$ ...

14 Complexité : diviser pour régner

On se ramène souvent au cas où n est de la forme $n = 2^k$ (donc $k = \log_2(n)$). À chaque fois on veut montrer, "il existe une constante $c > 0$ telle que, à partir d'un certain rang..."

- Tri fusion, Tri rapide : $C(n) = 2 \times C(\frac{n}{2}) + O(n)$.

  On résout $\begin{cases} C(2^k) \leq 2 \times C(2^{k-1}) + c.2^k \\ C(2^1) \leq 2.c \end{cases}$, et on montre par récurrence sur k que $C(2^k) \leq c.2^k \times k$ soit $C(n) = O(n.log_2(n))$

- Dichotomie récursive (comme il faut, en utilisant les indices) : $C(n) = 1 \times C(\frac{n}{2}) + O(1)$

  On résout $\begin{cases} C(2^k) \leq C(2^{k-1}) + c \\ C(2^1) \leq c \end{cases}$, et on montre par récurrence sur k que $C(2^k) \leq c.k$ soit $C(n) = O(log_2(n))$

- Dichotomie récursive (PAS comme il faut, avec du slicing) : $C(n) = 1 \times C(\frac{n}{2}) + O(n)$

  On résout $\begin{cases} C(2^k) = C(2^{k-1}) + c.\frac{2^k}{2} \\ C(2^0) = c \end{cases}$, et on montre par récurrence sur k que $C(2^k) = c.2^k$ soit $C(n) = O(n)$.

  (on a divisé par 2 parce que c'est ce qui se passe en réalité, et ça nous arrange pour les calculs)