# TD 2 : Feuille de préparation

## 1 Encore un effort sur les structures de données

**Exercice 1 :**

```
def racines_trinome(a, b, c) :
    d = b ** 2 - 4 * a * c
    if d < 0 : return( complex(-b, - pow(-d, 0.5)) / (2 * a), \
                                    complex(-b, pow(-d, 0.5)) / (2 * a) )
    if d == 0 : return(-b / (2 * a))
    if d > 0 : return((-b - pow(d, 0.5)) / (2 * a), (-b + pow(d, 0.5)) / (2 * a))
```

Cet algorithme n'est pas fonctionnel. En préciser la raison et proposer une correction.

**Exercice 2 :**
Petits casse-tête.

```
a, b, x = 4, 5, [1, 2]

def f(x):
    a, b = 1, 12
    x = x + 2
    def g(x):
        a, b = 3, 'a'
        print(a, b, x)
    g(x + 3)
    print(a, b, x)

f(x[b - a])
print(a, b, x)


L = [ [1, 2], [3, 4] ]
for val in L:
    val[0], val[1] = L[1][0], val[0]
print(L)
```

Préciser ce que renvoient ces deux programmes.

**Exercice 3 :**
Un programme à tester

```
def ma_fonction(n):
    nombre = ""
    while n != 0:
        nombre = str(n % 2) + nombre
        n = n // 2
    return(int(nombre))

def mon_autre_fonction(n):
    if n == 1:
        return(1)
    else:
        return(int(str(mon_autre_fonction(n // 2)) + str(n % 2)))

print(ma_fonction(14))
print(mon_autre_fonction(14))
```

Indiquer ce que renvoie ce programme. Est-il sûr que ces programmes renvoient le résultat attendu ?

## 2 Spécifications et preuves de programmes

**Exercice 4 :**
Prouver la terminaison de la boucle suivante en donnant un variant de boucle.

```
# n est un entier naturel
while n > 1 :
    if n % 2 == 0 :
        n = n // 2
    else :
        n = n + 1
```

## 3 Quatre algorithmes à connaître

**Exercice 5 :**
Écrire une version itérative, puis une version récursive pour calculer la factorielle d'un entier.

Montrer que votre algorithme termine et est correct.

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ (n-1)! \times n & \text{si } n \geqslant 1. \end{cases}$$

**Exercice 6 :**
Écrire une version itérative, puis une version récursive pour déterminer la racine d'une équation $f(x) = 0$ par la méthode de *dichotomie* (voir cours IPT de 1re année).

**Exercice 7** (Exponentiation rapide) **:**
Pour élever un nombre à la puissance $n$, il existe un algorithme bien plus performant que la méthode naïve consistant à multiplier ce nombre $(n-1)$ fois par lui-même : il s'agit de la méthode dite d'*exponentiation rapide*. Étant donnés un réel positif $a$ et un entier $n$, on remarque que :

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair}, \\ a \cdot \left(a^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair}. \end{cases}$$

Cet algorithme se programme naturellement par une fonction récursive. Programmer cette exponentiation rapide et comparer sa performance à celle de la méthode itérative classique.

**Exercice 8** (Algorithme de Ruffini-Horner) **:**
La méthode de Ruffini-Horner de recherche d'une valeur approchée de racine d'un polynôme est publiée à quelques années d'intervalle par Paolo Ruffini (1765-1822, italien) et par William George Horner (1786-1837, britannique) mais il semble bien que Horner n'ait pas eu connaissance des travaux de Ruffini. La méthode de Horner est ensuite popularisée par les mathématiciens De Morgan et J.R. Young.

Supposons que l'on souhaite calculer la valeur en $x_0$ du polynôme :

$$p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_n \cdot x^n$$

La première idée pour évaluer $p$ en $x_0$ est de calculer chaque puissance de $x_0$ de manière naïve, de multiplier par les coefficients, puis de tout additionner (ce qui nécessite $n + (n-1) + \ldots + 2 + 1 = n \cdot (n-1)/2$ produits).

On peut certes diminuer le nombre de multiplications à effectuer en utilisant l'algorithme d'exponentiation rapide.

La méthode de Horner permet de réduire encore le nombre de multiplications, en remarquant que :

$$p(x_0) = ((\ldots((a_n \cdot x_0 + a_{n-1}) \cdot x_0 + a_{n-2}) \cdot x_0 + \ldots) \cdot x_0 + a_1) \cdot x_0 + a_0$$

Ce faisant, il n'y a plus que $n$ multiplications à effectuer !

Pour programmer cet algorithme, il suffit de calculer les $n$ valeurs de la suite $b_n$ définie par :

$$\begin{cases} b_n = a_n \\ \forall\, k \in [\![0, n-1]\!],\ b_k = a_k + b_{k+1} \cdot x_0 \end{cases}$$

Programmer l'algorithme de Horner de manière récursive (le polynôme $p$ étant représenté par la liste de ses coefficients $[a_0, \ldots, a_n]$).