Exercice I : *CCP 2017 MP*

On peut proposer les solutions suivantes :

1.
```
A = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
```

2.
```
len(A) = 4
A[1] = [4,5,6]
A[2][1] = 8
```

3. Classiquement ou par compréhension
```
def difference(x,y):
    result = []
    for i in range(len(x)):
        result.append( x[i] - y[i]
        ↪  )
    return result
```
```
def difference(x,y):
    return [ x[i]-y[i] for i in range(len(x)) ]
```

4. Classiquement ou par compréhension
```
def norme(x):
    result = abs(x[0])
    for i in range(1,len(x)):
        if abs(x[i]) > result:
            result = abs(x[i])
    return result
```
```
def norme(x):
    return max( [ abs(xi) for xi in x ] )
```

5. Classiquement ou par compréhension
```
def itere(x,A):
    p = len(x)
    result = []
    for i in range(p):
        y = 0
        for j in range(p):
            y = y + x[j]*A[j][i]
        result.append(y)
    return result
```
```
def itere(x,A):
    p = len(x)
    return [ sum( [x[j]*A[j][i] for j in range(p)] )
    ↪  for i in range(p) ]
```

6.
```
def probainvariante(A,eps):
    p = len(A)
    mu0 =[ 1/p ]*p
    mu1 = itere( mu0 , A )
    while norme ( difference ( mu1 , mu0 ) ) > eps :
        mu0 = mu1
        mu1 = itere( mu0 , A )
    return mu1
```

Exercice II : *CCP 2019 MP*

☐ **Q1 -**

```
def est_premier(n : int) -> bool :
    if n < 2 : return False    # 0 ou 1 n'est pas premier
    if n == 2 : return True    # 2 est premier
    if n%2 == 0 : return False    # un nombre pair (sauf 2) n'est pas premier
    d = 3
```

```
    while d**2 <= n :
        if n%d == 0 :       # si d divise n
            return False    # n n'est pas premier
        d += 2
    return True
```

☐ **Q2 -**

```
def liste_premiers(n : int) -> list :
    if n<2 :
        return []
    lpremiers = [2]
    d = 3
    while d <= n :          # on cherche les nombres premiers parmi les nombres impairs <= n
        if est_premier(d) : # d est premier
            lpremiers += [d]
        d += 2              # prochain impair
    return lpremiers
```

☐ **Q3 -** On compte le nombre de divisions de n par p possibles.

```
def valuation_p_adique(n : int, p : int) -> int :
    v = 0
    while n%p == 0 :
        n = n//p
        v += 1
    return v
```

☐ **Q4 -** La condition d'arrêt : si p ne divise pas n, on renvoit 0.
L'hérédité : si p divise n, on compte 1 fois p et on ré-évalue la valuation p-adique sur $n//p$. On fera la somme de ce 1 avec la ré-évaluation p-adique.

```
def val(n : int, p : int) -> int :
    if n%p != 0 : # condition d'arrêt
        return 0
    else :
        return 1 + val(n//p, p) # hérédité
```

☐ **Q5 -**

```
def decomposition_facteurs_premiers(n : int) -> list :
    lpremiers = liste_premiers(n)
    fpremiers = []
    for p in lpremiers :
        vp = valuation_p_adique(n, p)
        if vp != 0 :
            fpremiers += [ [p, vp] ]
    return fpremiers
```

Exercice III : *e3a 2017 PSI*

Partie A : Recherche de zéro d'une fonction

1. (a) puisque f est continue, le théorème des valeurs intermédiaires permet de répondre par l'affirmative.

(b)
```
def recherche_dicho(f, a, b, eps):
    c = (a+b)/2
    while abs(b-a) > eps :
       if f(c) < 0 : a = c
       elif f(c) > 0 : b = c
       else : return c,c
       c = (a+b)/2
    return a,b
```

2. (a) Si $f(0) = 0$ ou si $f(1) = 1$ l'existence est assurée.
Traitons le dernier acs en supposant que $f(0) > 0$ et que $f(1) < 1$, puis considérons la fonction $g : x \mapsto f(x) - x$, par hypothèse sur f nous avons $g(0) > 0$ et $g(1) < 0$. Puisque g est continue sur [0,1], on peut appliquer le **Q1-a)** et ainsi $\exists c \in [0,1], g(c) = 0$. Autrement dit, f possède un point fixe en c.

(b)
```
def recherche_ptfixe(f, eps):
    if f(0)==0: return 0,0
    if f(1)==1: return 1,1
    return recherche_dicho(lambda x: f(x)-x, 0, 1, eps)
```

Partie B : Recherche dans une liste

1. (a) `recherche_dicho(L,1,5,6)` renvoie 3.
En effet,

a	1	1
b	5	3

et au final a=b=3.

`recherche_dicho(L,0,5,1)` renvoie 0.
En effet,

a	0	0	0
b	5	2	1

et au final a=b=0.

(b) Le programme `recherche_dicho` vérifie dans un premier temps si la valeur recherchée x est supérieure à toutes les valeurs comprises entre les indices g et d.
Une fois fait, il découpe en deux la sous-liste `L[g:d]` et regarde si la valeur milieu (le pivot) est inférieur à x. Selon ce test, on sait dans laquelle des deux sous-listes x peut s'insérer pour garder une liste ordonnée. L'algorithme met à jour les indices gauche et droit de la sous-liste dans laquelle x peut s'insérer et recommence jusqu'à ce que la nouvelle sous-liste soit vide. Il renvoie alors l'indice de cette sous-liste vide.

(c) Nous avons une comparaison au début puis deux par boucle. Ainsi si on note C la complexité de cet algorithme, au mieux $C = 1$ au pire $C = 1 + 2 \times log_2(d - g)$. Ainsi la complexité est en $\mathcal{O}(ln(d - g))$.

2
```
def tri_dicho(L):                  # L est une liste quelconque de nombres
    for k in range(1,len(L)):      # en fin de boucle, L[:k] sera triee
       x=L[k]
       p=recherche_dicho(L,0,k-1,x)
       for i in range(k,p,-1) :    # on decale les valeurs à partir de L[k]
          L[i]=L[i-1]
       L[p]=x
    return L
```

3. Lors du décalage des valeurs (petite boucle `for i`), il y a k-p affectations et dans le pire des cas (liste ordonnée du plus grand au plus petit) k affectations. On effectue n tours de boucle principale (où $n = len(L)$).
Ainsi le nombre d'affectations de `tri_dicho` est en $\mathcal{O}(n^2)\left(= \mathcal{O}\left(\sum_{k=1}^{n} k\right)\right)$. Le nombre de comparaisons de `tri_dicho` est n fois en $\mathcal{O}(ln(k))$ ainsi en $\mathcal{O}(nln(n))$.
Dans le pire des cas, tant pour le nombre d'affectations que le nombre de comparaisons, le tri par insertion classique est en $\mathcal{O}(n^2)$.
On préfèrera donc le tri par insertion dichotomique qui permet un léger gain.