Une correction possible du devoir n°1

Exercice 1

1. Première solution : imbriquer une boucle sur *x* in *L* dans une boucle sur *k* in range(*N*) et incrémenter *P*[*k*] chaque fois que *x* == *k*.
On peut utiliser la fonction sum et les listes en compréhension.

Complexité : $N \times \text{len}(L)$

```
def comptage(L,N):
    return [sum(1 for x in L if x == k) for k in
                                              range(N)]
```
ou
```
return [sum(x==k for x in L) for k in range(N)]
```

Deuxième solution (bien meilleure) : initialiser la liste *P* en la remplissant de 0 puis parcourir la liste *L* une seule fois.

Complexité : $N + \text{len}(L)$

```
def comptage(L,N):
    P=[0]*N
    for i in L:
        P[i] +=1
    return P
```

Dans la suite, nous utilisons cette solution.

2. Plusieurs solutions, par exemple :

```
def tri(L,N):
    P=comptage(L,N)
    M=[]
    for k in range(N):
        M.extend([k]*P[k])
    return M
```

```
def tri(L,N):
    P=comptage(L,N)
    return [k for k in range(N) for i in range(P[k])]
```

3. Pour évaluer la complexité de la fonction comptage, nous comptons les affectations/incrémentations.
La ligne P=[0]*N effectue *N* affectations. Ensuite, la boucle comporte *n* affectations, où $n = \text{len}(L)$.
La fonction tri réalise *n* affectations. La complexité de cet algorithme est donc $2n + N$.

Dans le cas particulier d'une liste déjà triée (ou presque), le tri par insertion a une complexité en $\Theta(n)$.
En moyenne et dans le pire des cas, le tri par insertion a une complexité en $\Theta(n^2)$, et le tri fusion toujours en $\Theta(n \ln n)$.

Cet algorithme ne sera pas performant dans le cas où *N* est beaucoup plus grand que *n* (tri des pays du monde par exemple, $n < 200$, $N > 1\,000\,000\,000$). Dans le cas où *N* est borné par une constante ou par une fonction affine de *n*, la complexité du tri défini dans l'exercice est en $\Theta(n)$. Il est donc plus performant que le tri par insertion (sauf si la liste est déjà triée ou presque) et que le tri fusion (dans tous les cas).

Complément : Ce tri est appelé tri comptage ou tri casier. Sa complexité est linéaire, donc il est plus rapide que les tris par comparaisons (complexité optimale en $\Theta(n \ln n)$), mais il ne fonctionne que sur des listes bornées d'objets assimilables à des entiers, et ne présente un intérêt que si les bornes ne sont pas trop éloignées.

Exercice 2

1.
```
def identite(n):
    M = [[0]*n for i in range(n)]
    for i in range(n):
        M[i][i]=1
    return M
```

```
def identite(n):
    return [[0 if j != i else 1 for j in range(n)] for i in range(n)]
```
ou
```
    return [[0]*i+[1]+[0]*(n-i) for i in range(n)]
```

2.
```
def determinant(M):
    n = len(M)
    if n == 2:
        return M[0][0]*M[1][1]-M[1][0]*M[0][1]
    else:
        det = 0
```

```
    for j in range(0,n):
        # Construction de la matrice extraite en supprimant la ligne 0 et la colonne j
        Mj=[]
        for i in range(0,n-1):
            Mj.append([M[i+1][k] for k in range(n) if k != j])
        det += (-1)**j*M[0][j]*determinant(Mj)
    return det
```

Autre solution

```
def determinant(M):
    n = len(M)
    if n == 2:
        return M[0][0]*M[1][1]-M[1][0]*M[0][1]
    else:
        return sum((-1)**j*M[0][j]*determinant(
            [M[1:][k][:j]+M[1:][k][j+1:] for k in
             range(n-1)]) for j in range(n))
```

M[1:] est la matrice obtenue en supprimant la première ligne. On parcourt toutes lignes de cette matrice en utilisant M[1:][k] for k in range(n-1), et on prend uniquement les colonnes 0 à j exclue et j+1 à n (exclue).

La bibliothèque numpy contient des objets matrices. Vous testerez :

```
import numpy as np
M=np.matrix([[1,2,3],[4,5,6],[7,8,10]]) ou M=np.matrix('1 2 3;4 5 6;7 8 10')
np.linalg.det(M)
```

Autre commandes intéressantes (à connaitre) : 3*M+M**(–1), np.linalg.eig(M) …

Pour effectuer une vraie copie de la matrice M, avant de supprimer la ligne 0 et la colonne j :

```
import copy
N=copy.deepcopy(M)
```

3. Soit d_n le nombre d'appels de la fonction *determinant* nécessaires au calcul du déterminant d'une matrice carrée M d'ordre n.

$d_2 = 1$ et si $n > 2$, $d_n = n \times d_{n-1}$, d'où pour $n \geq 2$, $d_n = \frac{n!}{2}$, et puisqu'il y a deux multiplications par appel, la fonction déterminant effectue $n!$ multiplications.

4. Le pivot de Gauss a une complexité en $O(n^3)$ (résultat de cours de première année, 3 boucles imbriquées) et sera donc plus rapide que la fonction *determinant*.
Par contre, la fonction *determinant* est plus fiable car elle n'effectue que des additions et des multiplications, alors que la fonction *gauss* effectue des divisions qui créent des erreurs d'arrondis.

5.

```
def sylvester(n):
    if n == 1:
        return [[1]]
    else:
        H=sylvester(n/2)
        m=len(H)
        H.extend([H[i][:] for i in range(m)])
        for i in range(m):
            H[i].extend([H[i][k] for k in range(m)])
            H[m+i].extend([-H[i][k] for k in range(m)])
        return H
```

On utilise H[i][:] au lieu de H[i] afin de créer une nouvelle liste contenant les mêmes valeurs que H[i]. Ainsi, les lignes de la matrice H forment des listes bien distinctes. Dans le cas contraire, différentes lignes pointeraient vers les mêmes listes et toute modification effectuée sur une ligne serait également effectuée sur ses copies.