```python
#Correction_info2

#EX1
def evaluation(L:list,z0:complex)->complex:
    """calcule l'image de z0 par P"""
    S=0
    for k in range(0,len(L)):
        S=S+L[k]*z0**k
    return S

def horner(L:list,z0:complex)->complex:
    """calcule l'image de z0 par P par l'algorithme de Hörner"""
    n=len(L)-1 #on définit le degré du polynôme pour faciliter les
notations
    u0=L[n]
    for k in range(1,n+1):
        u1=u0*z0+L[n-k]
        u0=u1 #une fois la valeur calculée, on la réinjecte pour le
tour suivant
    return u0

#EX2
def fibonacci(n:int)->int:
        """renvoie un de façon récursive"""
        if n==0 or n==1:
            return 1
        else:
            return fibonacci(n-1)+fibonacci(n-2)

def fibonacci2(n:int)->list:
        """renvoie un à l'aide d'une liste qui stocke les termes au
fur et à mesure"""
        if n==0:
            return [1]
        elif n==1:
            return [1,1]
        else:
            L=[1,1]
            for k in range(2,n+1):
                aux=L[k-2]+L[k-1]
                L.append(aux)
            return L

#EX3
#L'instruction pour connaître l'image de 1 est : s[1]. De la meme
façon, la transposition (2 3) est définie par ses images :
t=[0,1,3,2], en veillant à ce que 0 et 1 soient invariants.

def comp(s1:list,s2:list)->list:
    """renvoie la composition des permutations s1 et s2"""
    n=len(s1)
    p=len(s2)
    assert n==p
    s=[]
    for i in range(0,n):
```



```python
        s.append(s2[s1[i]])
    return s

def inv(s1:list)->list:
    """renvoie l'inverse de la permutation donnée"""
    n=len(s1)
    s=[0 for i in range(0,n)] #on construit une liste de n images
qu'on va compléter au fur et à mesure
    for i in range(0,n):
        s[s1[i]]=i
    return s

#EX4
def chercherdicoto(L:list,x:float)->bool:
    """cherche par dichotomie si un nombre x est dans une liste déjà
triée
    """
    n=len(L)
    a,b=0,n-1
    while a<=b:
        c=(a+b)//2  #la liste étant triée, on regarde le terme du
milieu.
        if x==L[c]:
            return True
        elif x<L[c]:
            b=c-1
        elif x>L[c]:
            a=c+1
    return False

#EX5
def presence(text:str,mot:str)->bool:
    """teste la présence du mot dans un texte"""
    #on va parcourir le texte jusqu'à repérer la première lettre du
mot, puis on vérifiera les lettres suivantes.
    n=len(mot)
    for k in range(0,len(text)-n+1): #il est inutile d'aller trop
loin..
        if text[k]==mot[0]:
            #on teste les autres lettres du mot, en utilisant par
exemple un compteur.
            c=0
            for i in range(0,n):
                if text[k+i]==mot[i]:
                    c=c+1
            if c==n:
                return True
        else:
            pass
    return False

def position(text:str,mot:str)->tuple:
    """teste la présence du mot dans un texte et renvoie la liste des
positions du mot"""
    #on adapte le programme précédent et on va stocker les positions
```

```
du mot dans une liste.
    n=len(mot)
    pos=[]
    for k in range(0,len(text)-n+1):
        if text[k]==mot[0]:
            #on teste les autres lettres du mot, en utilisant par
exemple un compteur.
            c=0
            for i in range(0,n):
                if text[k+i]==mot[i]:
                    c=c+1
            if c==n:
                pos.append(k)
        else:
            pass
    #on étudie alors la liste des positions après avoir parcouru le
texte
    if len(pos)==0:
        return False
    else:
        return True,pos
```