```python
##Exemple 2

def factorielle(n):
    '''calcul de la factorielle d'un entier naturel'''
    res=1
    for i in range(1,n+1):
        res=res*i
    return res   #nous avons trouvé une complexité linéaire

##Exemple 3
def factorielle_rec(n):
    '''calcul de n! en utilisant la récursivité'''
    if n==0 :
        return 1
    return n*factorielle_rec(n-1)

#Complexité: C(n)=2+C(n-1) , donc la suite (C(n)) est une suite arithmétique
    # de raison 2 et de premier terme C(0)=2
# On a donc C(n)=2+2n=O(n) complexité linéaire

##Exemple 4
def estpremier(n):
    '''test de primalité pour un entier n non nul'''
    if n==1:
        return(False)    #1 n'est pas premier
    for i in range(2,n): #on parcourt les entiers de 2 à n-1
        if n%i==0:                      # on teste si i est un diviseur de n
            return False   # on sort de la boucle car il est inutile de voir si n
            admet d'autres diviseurs
    return True          # si on n'est pas sorti de la boucle plus tôt, cela signifie
    que n est premier

#la complexité est constante dans le meilleur des cas (quand n est pair),
#☺   linéaire dans le pire des cas (quand n est premier)


##Exercice 2
def somme(n):
    '''calcul de la somme(k=0..n;1/k!) en utilisant la fonction factorielle'''
    s=1 #initialisation avec la valeur k=0
    for k in range(1,n+1):
        s=s+1/factorielle(k)
    return s   #nous avons trouvé une complexité quadratique

def somme_amelioree(n):
    '''calcul de la somme(k=0..n;1/k!)  sans utiliser la fonction factorielle'''
    s=1
    f=1  #ppour stocker les valeurs successives de k!
    for i in range(1,n+1):
        f=f*i
        s=s+1/f
    return s #nous avons trouvé une complexité linéaire


## Exercice 3
def est_present(L,x):
    for elt in L:
        if elt==x:
            return True
    return False
#complexité linéaire (dans le pire des cas la boucle s'execute n fois)
```

```python

## Exercice 4
def recherche_dicho(L,x):
    '''la liste L doit être triée par ordre croissant'''
    n=len(L)
    mini=0
    maxi=n-1

    while mini <= maxi:
        milieu=(mini+maxi)//2
        if x==L[milieu]:
            return True
        if x<L[milieu]:
            maxi=milieu-1
        if x>L[milieu]:
            mini=milieu+1
    return False

#complexité: pour une liste de longueur 2^p, la boucle va s'executer au maximum p fois
    # car à chaque tour on cherche dans une liste de taille divisée par 2
# donc si n=2^p , C(n)=C(2^p)=4+6*p=4+6*log_2(n) (logarithme en base 2)
# On a donc une complexité logarithmique

##par récursivité
def recherche_dicho_rec(L,x):
    '''la liste L doit être triée par ordre croissant'''
    n=len(L)
    if n==0:
        return False

    milieu=(n-1)//2
    if x==L[milieu]:
        return True
    if x<L[milieu]:
        return recherche_dicho_rec(L[:milieu],x)
    if x>L[milieu]:
        return recherche_dicho_rec(L[milieu+1:],x)

#complexité (plus dur): C(2^p)=10+C(2^(p-1))
        # donc si on pose u_p=C(2^p), alors (u_p) est une suite arithmétique de raison 10
        # donc u_p=2+10p
        # donc C(n)=C(2^p)=2+10p=2+10log_2(n)
# On a donc une complexité logarithmique
```