Problème 1 : produit de polynômes

1. a) La liste étant non vide, il suffit de prévoir le cas du polynôme nul

```
1  def deg(p) :
2      if p == [0] :
3          return -1
4      else :
5          a = p[-1]
6          if a !=0 :
7              return len(p)-1
8          else :
9              p.pop()
10             return deg(p)
```

b) La méthode `pop()` supprime les zéros inutiles donc `p` va devenir `[0,1,0,2]`.

2. a) Inutile de supprimer les zéros qui pourraient apparaître en tête de la liste :

```
1  def sum(p,q) :
2      r = []
3      for i in range(len(p)) :
4          r.append(p[i]+q[i])
5      return r
```

b) Là encore, inutile de faire un cas particulier avec $a = 0$:

```
1  def prodScal(p,a) :
2      r = []
3      for k in p :
4          r.append(a*k)
5      return r
```

c) On rajoute le bon nombre de zéros au début de la liste :

```
1  def prodMonome(p,r) :
2      return [0]*r + p
```

3. a) Si $P, Q \in \mathbb{R}_n[X]$ alors $\deg(PQ) \leqslant 2n$ donc PQ devra être représenté par une liste de longueur $2n+1$.

b) Il faut penser à rajouter des zéros avant de pouvoir utiliser `sum` de façon à ce que toutes les listes soient de longueur $2n+1$

```
1  def mult(p,q) :
2      n = len(p)-1
3      r = [0]*(2*n+1)
4      for k in range(n+1) :
5          s = prodMonome(q,k)
6          s = prodScal(s,p[k])
7          s = s+[0]*(n-k)
8          r = sum(r,s)
9      return r
```

c) Seule la fonction `prodScal` fait des multiplications, elle en fait $n+1$ à chaque appel et est appelée $n+1$ fois donc le nombre de multiplication est $(n+1)^2$.

4. a) `r` et `q` sont le début et la fin de la liste `p` mais il faut rajouter un coefficient nul en tête de `r` pour le faire apparaître comme une liste de longueur $\frac{1}{2}n+1$ alors que R n'est que de degré $\leqslant \frac{1}{2}n-1$:

```
1  def decompose(p) :
2      n = (len(p)-1)//2
3      return p[n:],p[:n]+[0]
```

b) Il suffit de développer !

c) On calcule les produits Q_1Q_2, $(Q_1+R_1)(Q_2+R_2)$ et R_1R_2 récursivement ; il faut faire attention que le cas de base est obtenu pour $k=0$ donc $n=2^0=1$ (la fonction présentée ici ne fonctionne pas pour les polynômes constants, il faudrait prévoir un cas supplémentaire pour cela). Il faut également faire attention pour utiliser la fonction `sum` qui a été codée pour des polynômes de même degré, il faut donc rajouter des zéros en tête des polynômes si ce n'est pas le cas :

```
def prod(p,q) :
    if len(p) == 2 :
        return [p[0]*q[0],p[0]*q[1]+p[1]*q[0],p[1]*q[1]]
    else :
        n = len(p)-1
        q1,r1 = decompose(p)
        q2,r2 = decompose(q)
        q1q2 = prod(q1,q2)
        r1r2 = prod(r1,r2)
        s1 = sum(q1,r1)
        s2 = sum(q2,r2)
        res = prod(s1,s2)
        res = sum(res,prodScal(q1q2,-1))
        res = sum(res,prodScal(r1r2,-1))
        res = prodMonome(res,n//2)
        res = sum(res,r1r2+[0]*(n//2))
        res = sum(res+[0]*(n//2),prodMonome(q1q2,n))
        return res
```

Pour un polynôme de degré $n = 2^k$, on fait 3 appels récursifs à la fonction et 2 utilisations de la fonction `prodScal` (que l'on pourrait négliger vu qu'ils ne font que changer des signes) à des polynômes de degré n (Q_1Q_2 et R_1R_2 sont de « degré » n). Si on note $C(k)$ le nombre de multiplications réalisées pour un polynôme de degré $n = 2^k$, on a donc $C(k) = 3C(k-1) + 2 \times 2^k$. On en déduit $C(k) + 2^{k+2} = 3\left(C(k-1) + 2^{k+1}\right)$ donc $C(k) + 2^{k+2} = 3^k\left(C(0) + 2^2\right)$ et comme $C(0) = 4$, on en déduit $C(k) = 8 \times 3^k - 2^{k+2} = O\left(3^k\right)$. Comme $k = \lfloor \log_2(n) \rfloor$, on en déduit $C(n) = O\left(n^{\ln(3)/\ln(2)}\right)$ puisque $3^{\ln(n)/\ln(2)} = \exp\left(\dfrac{\ln(n)}{\ln(2)}\ln(3)\right) = n^{\ln(3)/\ln(2)}$. On a effectivement un produit plus rapide puisque $\dfrac{\ln(3)}{\ln(2)} < 2$.

Problème 2 : algorithme de Kaprekar

1. a) On obtient

k	n	r	L
0	12	3	[3]
1	1	2	[3,2]
2	0	1	[3,2,1]

`f(123,3)` renvoie `[3,2,1]`

k	n	r	L
0	2	2	[2]
1	0	2	[2,2]
2	0	0	[2,2,0]

`f(22,3)` renvoie `[2,2,0]`

b) La fonction `f` renvoie la liste des `p` chiffres de `n`

2. Un seul balayage de `L` suffit

```
def occ(L) :
    F = [0]*10
    for k in L :
        F[k] += 1
    return F
```

3. a) Les chiffres de `n` sont donc (dans un certain ordre) 0,0,2,4,4,5,7,7,7,8 donc `M=8777544200` et `m=24457778`

b) On peut le faire directement (la variable `s` permet de savoir à quel chiffre on en est pour ajuster la puissance de 10 à considérer)

```
def K(n,p) :
    L = occ(f(n,p))
    M,m = 0,0
    s = 0
    for k in range(10) :
        for i in range(L[k]) : # boucle vide si L[k]=0
            M += k*10**s
            m += k*10**(p-s-1)
            s += 1
    return M,m
```

On peut aussi le faire avec des chaines de caractères qui rendent les concaténation plus faciles puis les convertir en entiers à la fin

```
def Kbis(n,p) :
    L = occ(f(n,p))
    M,m = '','' 
    for k in range(10) :
        for i in range(L[k]):
            M = str(k)*L[k]+M # sans effet si L[k]=0
            m = m+str(k)*L[k]
    return int(M),int(m)
```

4. Il faut penser à rajouter la première répétition dans la liste avant de la renvoyer

```
def Kaprekar(n,p) :
    L = [n]
    M,m = K(n,p)
    while M-m not in L :
        n = M-m
        L.append(n)
        M,m = K(n,p)
    L.append(M-m)
    return L
```