I Binôme et plus

1. On remplit tout le triangle de Pascal de la ligne 0 à la ligne n avant d'extraire le coefficient souhaité :

```
def b(n,p) :
    Tr = [[0 for j in range(n+1)] for i in range(n+1)]
    Tr[0][0] = 1
    for i in range(1,n+1) :
        Tr[i][0] = 1
        for j in range(1,n+1) :
            Tr[i][j] = Tr[i-1][j-1]+Tr[i-1][j]
    return Tr[n][p]
```

2. a) Avec un seul `return` pour simplifier la suite :

```
def T(i,j,k) :
    n = i+j+k
    if i == 0 :
        r = b(n,j)
    elif j*k == 0 :
        r = b(n,i)
    else :
        r = T(i-1,j,k)+T(i,j-1,k)+T(i,j,k-1)
    return r
```

b) Avec deux fonctions :

```
def T1(i,j,k) :
    d = {}
    return t1(i,j,k,d)

def t1(i,j,k,d) :
    if (i,j,k) not in d :
        n = i+j+k
        if i == 0 :
            r = b(n,j)
        elif j*k == 0 :
            r = b(n,i)
        else :
            r = t1(i-1,j,k,d)+t1(
                i,j-1,k,d)+t1(i,j,
                k-1,d)
        d[(i,j,k)] = r
    return d[(i,j,k)]
```

Ou une sous-fonction :

```
def T2(i,j,k) :
    d = {}
    def g(i,j,k) :
        if (i,j,k) not in d :
            n = i+j+k
            if i == 0 :
                r = b(n,j)
            elif j*k == 0 :
                r = b(n,i)
            else :
                r = g(i-1,j,k)+g(
                    i,j-1,k)+g(i,j
                    ,k-1)
            d[(i,j,k)] = r
        return d[(i,j,k)]
    return g(i,j,k)
```

c) On remplit la pyramide de Pascal, étage par étage : c'est n qui désigne l'étage et on aura toujours $k = n-(i+j)$:

```
def T3(i,j,k) :
    n = i+j+k
    Pyr = [[[0 for y in range(n+1)] for x in range(n+1)] for h in range(n
        +1)]
    Pyr[0][0][0] = 1
    for h in range(1,n+1) :
        for x in range(h+1) :
            Pyr[h][x][0] = b(h,x)
            Pyr[h][0][x] = b(h,x)
        for x in range(1,h+1) :
            for y in range(1,h+1) :
                Pyr[h][x][y] = Pyr[h-1][x][y]+Pyr[h-1][x-1][y]+Pyr[h-1][x][
                    y-1]
    return Pyr[n][i][j]
```

II Suite de Syracuse

1.
```
def suivant(k) :
    if k%2 == 0 :
        return k//2
    else :
        return 3*k+1
```

2.
```
def tv(k) :
    if k == 1 :
        r = 0
    else :
        r = 1+tv(suivant(k))
    return r
```

3. Avec mémoïsation :

```
def tvMax(n) :
    d = {}
    def f(k) :
        if k not in d :
            if k == 1 :
                r = 0
            else :
                r = 1+f(suivant(k))
            d[k] = r
        return d[k]
    for k in range(1,n+1) :
        f(k)
    return max(d.values())
```

4. La suite des entiers depuis 3 est : 3,10,5,16,8,4,2,1 donc le temps de vol est 7 mais il est difficile de prévoir à l'avance par quelle valeur maximale va passer cette succession d'entier, ce qui serait nécessaire pour prédire la taille de la liste à introduire. On pourrait toujours commencer avec une liste de longueur donnée et la « ralonger » par concaténation en cas de dépassement.

III Optimiser un produit matriciel

1. Mémoïsée avec une sous-fonction :

```
def c(n) :
    d = {}
    def f(k) :
        if k not in d :
            if k in [1,2] :
                r = 1
            else :
                r = 0
                for i in range(1,k) :
                    r += f(i)*f(k-i)
            d[k] = r
        return d[k]
    return f(n)
```

2. Les notations ne sont pas pratiques, j'aurais plutôt dû numéroter les matrices de 0 à $k-1$ ...

a) On peut couper un tuple par slicing :

```
def N(t) :
    k = len(t)-1
    L = []
    if k <= 1 :
        r = 0
    else :
        L = [0 for _ in range(k-1)]
```

```
        for i in range(1,k) :
            t1,t2 = t[:i+1],t[i:]
            L[i-1] = N(t1)+N(t2)+t[0]*t[i]*t[-1]
        r = min(L)
    return r
```

b) Avec une sous-fonction :

```
def N_memo(t) :
    d = {}
    def f(t) :
        if t not in d :
            k = len(t)-1
            L = []
            if k <= 1 :
                r = 0
            else :
                L = [0 for _ in range(k-1)]
                for i in range(1,k) :
                    t1,t2 = t[:i+1],t[i:]
                    L[i-1] = N_memo(t1)+N_memo(t2)+t[0]*t[i]*t[-1]
                r = min(L)
            d[t] = r
        return d[t]
    return f(t)
```

c) C'est là que le décalage entre les indices dans le tableau (à partir de 0) et dans le produit de matrices (à partir de 1) rend les choses plus pénibles.

Il faut remplir de tableau en faisant décroître i (et croître j) ; le résultat final est dans la dernière case de la première ligne ($i = 0$, $j = k$). On a $i < j$ donc seule la « moitié » du tableau est utile.

```
def N_iter(t) :
    k = len(t)-1
    T = [[0 for j in range(k)] for i in range(k)]
    for i in range(k-2,-1,-1) :
        for j in range(i+1,k) :
            L = []
            for h in range(i+1,j+1) :
                L.append(T[i][h-1]+T[h][j]+t[i]*t[h]*t[j+1])
            T[i][j] = min(L)
    return T[0][-1]
```

L'initialisation de `T` et en $O(k^2)$ puis dans la double boucle imbriquée, la construction de `L` et le calcul de son maximum donne une complexité en $O(k^3)$.

d) Il faut arriver à reconstruire les étapes qui ont amené à la valeur précédente. Si on ne connaît que la tableau `T`, on va devoir refaire tous les calculs pour retrouver le chemin. Pour éviter cela, on remplit en même temps que `T`, un tableau `T1` pour lequel `T1[i][j]` correspond à la valeur de i qui a donné le minimum dans le calcul précédent (obtenu par la fonction `indiceMax`). Avec le tableau `T1`, la fonction récursive `afficher` permet de reconstruire le produit optimal :

```
def indiceMin(L) :
    ind = 0
    for i in range(len(L)) :
        if L[i] < L[ind] :
            ind = i
    return ind
```

```
def ordre(t) :
    k = len(t)-1
    T = [[0 for j in range(k)] for i in range(k)]
    T1 = [[0 for j in range(k)] for i in range(k)]
    for i in range(k-2,-1,-1) :
        for j in range(i+1,k) :
            L = []
            for h in range(i+1,j+1) :
                L.append(T[i][h-1]+T[h][j]+t[i]*t[h]*t[j+1])
            T[i][j] = min(L)
            T1[i][j] = i+indiceMin(L)     # c'est le numéro, compté dans le produit initial (depuis la matrice A1) du produit à effectuer en premier pour calculer Ai...Aj
    def afficher(T1,i,j) :
        if i == j :
            return 'A'+str(i+1)
        else :
            return '('+afficher(T1,i,T1[i][j])+afficher(T1,1+T1[i][j],j)+')'
    print(afficher(T1,0,k-1))
```

Un exemple d'utilisation :

```
>>> t = (2,3,12,7,4)

>>> N_memo(t)
296

>>> ordre(t)
(((A1A2)A3)A4)
```