# I Binôme et plus

**1.** On crée un tableau `Tr` suffisamment grand pour contenir les $n$ premières lignes du triangle de Pascal et on le remplit ligne par ligne

```
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) Tant qu'on n'atteint pas un « bord », on a $T(n) = 3T(n-1)$ donc la complexité du calcul de $T(n,n,n)$ est au moins en $3^n$.

b) Avec deux fonctions différentes ou une sous fonction

```
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)]
```

```
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) Même principe que pour les coefficients binomiaux :

```
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 Monter les escaliers

**1.** On commence soit par une marche, il en reste $n-1$ à monter, ce qui se fait de $u_{n-1}$ façons différentes, soit 2, soit 3 soit 4 donc $u_n = u_{n-1} + u_{n-2} + u_{n-3} + u_{n-4}$.

**2.** On mémoïse avec un dictionnaire et une sous fonction (par exemple)

```
def U(n) :
    d = {}
    def u(n) :
        if n not in d :
            if n <= 4 :
                r = 2**(n-1)
            else :
                r = u(n-1)+u(n-2)+u(n-3)+u(n-4)
            d[n] = r
        return d[n]
    return u(n)
```

**3.** Une liste de longueur $n$ permet de mémoriser toutes les valeurs de $u_1$ à $u_n$ : on met les 4 premiers éléments pour éviter des tests sur la valeur de $n$ ; cela ne pose pas de problème si $n < 4$ car la deuxième liste dans la concaténation est vide dans ce cas, tout comme la boucle `for`

```
def U_iter(n) :
    L = [1,2,4,9] + [0]*(n-4)
    for i in range(4,n) :
        L[i] = L[i-1]+L[i-2]+L[i-3]+L[i-4]
    return L[n-1]
```

Complexité linéaire

# III Chemins dans un carré

**1.** Si on n'est pas sur le bord, ie $x < N$ et $y < N$, on fait soir un pas vers la droite et il reste à aller de $(x+1, y)$ à $(N, N)$, soit un pas vers le haut et on sera en $(x, y+1)$, soit un pas en diagonale pour aller en $(x+1, y+1)$. Si $x = N$, on ne peut plus aller que vers le haut (il n'y a en fait qu'un seul chemin dans ce cas) ; si $y = N$, on ne peut aller que vers la droite.

$$f(x,y) = \begin{cases} f(x+1,y) + f(x,y+1) + f(x+1,y+1) & \text{si } x < N \text{ et } y < N \\ f(x+1,y) & \text{si } y = N \text{ et } x < N \\ f(x,y+1) & \text{si } x = N \text{ et } y < N \\ 1 & \text{si } x = y = N \end{cases}$$

**2.** Soit en version récursive

```
def chemins(N) :
    d = {}
    return Chemins(0,0,N,d)

def Chemins(x,y,N,d) :
    if (x,y) not in d :
        if x == N and y == N :
            r = 1
        elif x == N :
            r = Chemins(x,y+1,N,d)
        elif y == N :
            r = Chemins(x+1,y,N,d)
        else :
            r = Chemins(x+1,y,N,d)+Chemins(x,y+1,N,d)+Chemins(x+1,y+1,N,d)
        d[(x,y)] = r
    return d[(x,y)]
```

Ou de bas en haut : on remplit un tableau de taille $(N+1) \times (N+1)$ depuis la case $(N, N)$, en « remontant » ; `T[i][j]` contient la valeur de $f(i, j)$ donc la valeur cherchée au final est `T[0][0]`. On voit facilement la complexité quadratique par cette méthode.

```
def chemins_iter(N) :
    T = [[1 for j in range(N+1)] for i in range(N+1)]
    for i in range(1,N+1) :
        for j in range(1,N+1) :
            T[N-i][N-j]=T[N-i+1][N-j]+T[N-i][N-j+1]+T[N-i+1][N-j+1]
    return T[0][0]
```