# Devoir en temps libre d'informatique n° 01

Vous numéroterez vos copies et ferez apparaître clairement sur la première page le nombre de copies. Vous prêterez une attention particulière **au soin** de vos copies.
Si vous avez accès à un ordinateur muni de Python (environnement de base ou plus évolué), n'hésitez pas à tester vos fonctions.

## 1 Exercice : Tableaux binaires

Dans cet exercice, les structures de données étudiées sont des tableaux carrés dont les coefficients sont uniquement des 0 et des 1 (on parlera de tableaux binaires). En Python, on pourra considérer un tel tableau comme une liste de listes .
**Par exemple**, l'objet $T = [[1,0,1,1],[0,0,0,0],[1,1,0,0],[0,0,0,1]]$ représente le tableau :

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

L'accès à l'élément de la ligne $i$ et de la colonne $j$ s'obtient par la syntaxe : $T[i][j]$.

1. . Soit $N \in \mathbb{N}^*$. Combien existe-t-il de tableaux binaires distincts de taille $N \times N$ ? En considérant que les entiers sont codés sur 32 bits, donner l'ordre de grandeur (en Go) de l'espace mémoire nécessaire pour stocker tous les tableaux de taille $6 \times 6$.

   On dit qu'un tableau binaire est *équilibré* lorsqu'il y a autant de 0 que de 1. On dit qu'un tableau binaire est *totalement déséquilibré* lorsqu'aucun sous-tableau carré que l'on peut former de façon contigue en partant du coin supérieur gauche n'est *équilibré*.
   **Exemple**. Parmi les tableaux suivants, $T_e$ est équilibré, $T_d$ est totalement déséquilibré et le tableau $T_r$ n'est ni l'un ni l'autre :

$$T_e = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \quad , \quad T_d = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{et} \quad T_r = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   Pour $T_d$, on vérifie en effet qu'aucun des sous-tableaux suivants n'est équilibré :

$$(1) \quad , \quad \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad , \quad \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad , \quad \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   Pour simplifier, on supposera désormais que le nombre de lignes (et donc de colonnes) est pair.
2. Définir une fonction Python **compt0(T)** dont l'argument est un tableau binaire $T$ et qui retourne le nombre de 0 dans le tableau.
3. Définir une fonction Python **equilibre(T)** dont l'argument est un tableau binaire $T$ et qui retourne le booléen **True** si le tableau $T$ est équilibré et **False** sinon.

   Pour un tableau binaire $T$, on définit la mesure d'équilibre **mes(T)** comme étant le plus grand entier $k$, s'il existe, tel que le sous-tableau carré de $T$ de taille $2k \times 2k$ extrait en partant du coin supérieur gauche est équilibré ; si un tel entier n'existe pas, on pose mes(T) = 0

   **Exemple**. $mes(T_e) = 2, mes(T_d) = 0$ et $mes(T_r) = 1$.
4. Définir une fonction Python **mes(T)** qui réalise le travail demandé.

5. Définir une fonction Python **desequilibre(T)** dont l'argument est un tableau binaire $T$ et qui retourne le booléen **True** si le tableau $T$ est totalement déséquilibré et **False** sinon.
6. **Simulations et estimations de fréquences**.
   (a) A l'aide de la fonction **randrange** du module **random** et qui donne un nombre entier aléatoire dans l'intervalle entier $[a, b[$, écrire une fonction **alea(N)** qui retourne un tableau binaire $N \times N$ dont les coefficients sont pris aléatoirement
   (b) Pour $N = 4, 6, 8, \cdots$ estimer les proportions de tableaux équilibrés et totalement déséquilibrés en comptant le nombre d'occurrences de tels tableaux sur un grand nombre de tableaux pris au hasard ; les estimations seront réalisés à l'aide d'un échantillon suffisamment grand (au moins 1000 tableaux). Quelle est la fréquence théorique d'apparition d'un tableau binaire équilibré de taille $N \times N$ ?

# 2 Problême : Factorielle et coefficients binomiaux

## 2.1 Partie I : Factorielle

On considère l'algorithme suivant

```
>>> f = 1
>>> for i in range(1, n+1):    # n étant un entier naturel donné
>>>         f = f * i
```

1. Suivre l'état des variables $i$ et $f$ dans cet algorithme pour l'entrée $n = 6$.
2. Déterminer un *invariant de boucle* permettant de justifier que l'algorithme précédent retourne bien $n!$ lorsque $n$ est un entier strictement positif
3. Calculer le *nombre de multiplications* d'entiers pour une entrée $n$ donnée
4. Écrire une fonction Python **factorielle(n)** qui reprend l'algorithme ci-dessus. On veillera cependant à ce que l'appel factorielle(0) retourne 1.

## 2.2 Partie II : Calcul d'un coefficient binomial

1. Voici une fonction **binomial1(n,k)** pour le calcul des coefficients binomiaux :

   ```
   >>> def binomial1(n, k):
   >>>         return (factorielle(n) // (factorielle(k) * factorielle(n-k))
   ```

   Calculer le nombre de multiplications d'entiers de l'appel de la fonction **binomial1(n,k)**. En déduire que la complexité en nombre de multiplications est en $O(n)$
2. (a) Question mathématique Montrer les identités suivantes :

$$\binom{n}{k} = \frac{\prod_{i=0}^{k-1}(n-i)}{\prod_{i=0}^{k-1}(i+1)} \qquad \text{et} \qquad \binom{n}{k} = \binom{n}{n-k}$$

   (b) En déduire une fonction **binomial2(n,k)** pour le calcul des coefficients binomiaux nécessitant *moins* de multiplications que l'algorithme précédent. Majorer le nombre de multiplications d'entiers.

(c) Implémenter les fonctions **binomial1(n,k)** et **binomial2(n,k)** en Python et comparer leur vitesse à l'aide de la fonction **time** du module **time** vue en TP. On pourra réaliser un tableau comparatif pour plusieurs grandes valeurs de $n$ et $k$ afin de mettre en évidence une différence significative entre les fonctions.

3. Remarquant que $\binom{n}{k} = \prod_{i=0}^{k-1} \left(\frac{n-i}{i+1}\right)$, un élève définit alors la fonction **binomial3(n,k)** suivante :

```
>>> def binomial3(n, k):
>>>         prod = 1
>>>         for i in range(0,k):
>>>                 prod = prod * (n-i)/(i+1)
>>>         return (int(prod))
```

Il obtient alors les résultats contradictoires :

```
>>> binomial2(59, 22)
8964377427999630
```

```
>>> binomial3(59, 22)
8964377427999631
```

(a) Question mathématique Montrer que si $n$ est un entier inférieur á 124 ($= 5^3 - 1$), alors la valuation 5-adique de $n!$ est égal à : $\lfloor \frac{n}{5} \rfloor + \lfloor \frac{n}{25} \rfloor$

(b) En déduire que le résultat de binomial3(59, 22) est faux. Comment expliquer cette erreur ? Corriger la section de la page Wikipedia à l'adresse [1] :

`en.wikipedia.org/wiki/Binomial_coefficient`

## 2.3 Partie III : Calcul de tous les coefficients binomiaux

De nombreux problèmes nécessitent d'avoir accès à tous les coefficients binomiaux (ou au moins à ceux d'une ligne du triangle de Pascal).

(a) On fixe un entier $N$. Par un calcul direct utilisant une des deux fonctions **binomial** de la partie précédente, donner le nombre de calculs nécessaires à l'obtention de tous les coefficients binomiaux $\binom{N}{k}$ pour $k$ allant de 0 à $N$. Quelle est la complexité d'une telle méthode ?

Dans ce genre de situation, il est plus sage de faire appel à la relation de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

On veut concevoir une fonction Python **tableauBinomial(N)** retournant la liste des listes $\left[\binom{n}{k} \text{ pour } k = 0 \cdots N\right]$ avec $n$ variant entre 0 et $N$. Ainsi si $T = tableauBinomial(N)$ alors T[n][k] $= \binom{n}{k}$.

Par exemple, pour $N = 3$, on doit trouver la liste : [[1,0,0,0], [1,1,0,0], [1,2,1,0], [1,3,3,1]]

(b) Ecrire la fonction **tableauBinomial(N)**. Vous pouvez suivre la démarche indiquée ci-dessous :

☞ On initialise une variable locale tableau = [[0 for j in range(0,N+1)] for i in range(0,N+1)]

☞ On remplit la *première colonne* par les affectations tableau[i][0] = 1

1. Erreur encore visible le 15/02/2015

☞ On remplit itérativement le reste du tableau en utilisant la relation de Pascal et les affectations :
tableau[i][j] = tableau[i-1][j-1] + tableau[i-1][j]
Tester la fonction pour plusieurs valeurs de $N$

(c) Calculer le nombre d'opérations arithmétiques (additions) pour l'entrée $N$
Pour un affichage plus sympathique, passer la liste des coefficients binomiaux en argument de la procédure donnée ci-dessous.

```
>>> def affiche(T):
>>>         tab ='␣'
>>>         for ligne in T:
>>>                         for coeff in ligne :
>>>                                 tab += '␣' + str(coeff).center(5) + '␣'
>>>                         tab = tab + '\n'
>>>         print(tab)
```

(d) Modifier la procédure affiche(T) pour qu'elle affiche le caractère * si le coefficient binomial est impair et le caractère *espace* si le coefficient binomial est pair.

# 3 Exercice : Tableaux binaires

1. . Soit $N \in \mathbb{N}^*$. Pour chaque case, il y a deux choix possibles pour le coefficient : 0 ou 1. Comme il y a $N^2$ cases, on a $2^{N^2}$ **tableaux binaires distincts de taille** $N \times N$
Si les entiers sont codés sur 32 bits, un tableau binaire de taille $6 \times 6$ est codé sur $36 \times 4 = 144$ octets. Donc les $2^{36} = 68719476736 = 68,72 \times 10^9$ tableaux binaires de taille $6 \times 6$ nécessitent environ **9200 Go de mémoire**.

Évidemment, en effectuant une compression, en utilisant pleinement la structure des tableaux binaires et en codant chaque coefficient du tableau sur un bit, on peut coder un tableau sur 2 octets (et plus sûrement sur 3 octets car il faut alors préciser la structure de la liste...), on peut obtenir tous les tableaux binaires de taille $6 \times 6$ sur près de 200 Go... nombre qu'il faut multiplier par 500 millions pour obtenir tous les tableaux binaires de taille $8 \times 8$

On dit qu'un tableau binaire est *équilibré* lorsqu'il y a autant de 0 que de 1. On dit qu'un tableau binaire est *totalement déséquilibré* lorsqu'aucun sous-tableau carré que l'on peut former de façon contigue en partant du coin supérieur gauche n'est *équilibré*.
**Exemple**. Parmi les tableaux suivants, $T_e$ est équilibré, $T_d$ est totalement déséquilibré et le tableau $T_r$ n'est ni l'un ni l'autre : Pour simplifier, on supposera désormais que le nombre de lignes (et donc de colonnes) est pair.

2. Définir une fonction Python **compt0(T)** dont l'argument est un tableau binaire $T$ et qui retourne le nombre de 0 dans le tableau.

```
>>> def compt0(T):
>>>     N = len(T)
>>>     compte = 0
>>>     for k in range(N):
>>>         for j in range(N):
>>>             if T[j][k] == 0:
>>>                 compte += 1
>>>     return(compte)
```

3. Définir une fonction Python **equilibre(T)** dont l'argument est un tableau binaire $T$ et qui retourne le booléen **True** si le tableau $T$ est équilibré et **False** sinon.

```
>>> def equilibre(T):
>>>     N = len(T)
>>>     return (2*compt0(T) == N**2)
```

Pour un tableau binaire $T$, on définit la mesure d'équilibre **mes(T)** comme étant le plus grand entier $k$, s'il existe, tel que le sous-tableau carré de $T$ de taille $2k \times 2k$ extrait en partant du coin supérieur gauche est équilibré ; si un tel entier n'existe pas, on pose mes(T) $= 0$

4. Définir une fonction Python **mes(T)** qui réalise le travail demandé.

```
>>> def mes(T):
>>>     N = len(T)
>>>     for k in range(N//2,0,-1):
>>>         if equilibre([T[j][:2*k] for j in range(2*k)]):
>>>             return(k)
>>>     return(0)
```

5. Définir une fonction Python **desequilibre(T)** dont l'argument est un tableau binaire $T$ et qui retourne le booléen **True** si le tableau $T$ est totalement déséquilibré et **False** sinon.

```
>>> def desequilibre(T):
>>>         return (mes(T) == 0)
```

6. **Simulations et estimations de fréquences.**

(a) A l'aide de la fonction **randrange** du module **random** et qui donne un nombre entier aléatoire dans l'intervalle entier $[a, b[$, écrire une fonction **alea(N)** qui retourne un tableau binaire $N \times N$ dont les coefficients sont pris aléatoirement. On importe le module random avec l'alias rd

```
>>> def alea(N):
>>>     T = [ [1 for k in range(N)] for i in range(N)]
>>>     for k in range(N):
>>>         for j in range(N):
>>>             T[k][j] = rd.randrange(0,2)
>>>     return(T)
```

(b) Pour $N = 4, 6, 8, \cdots$, on effectue les simulations suivantes :

```
>>> nb = 1000
>>> for N in range(4,28,2):
>>>     cpteq, cptdes = 0, 0
>>>     for k in range(nb):
>>>         T = alea(N)
>>>         if equilibre(T):
>>>             cpteq += 1
>>>         if desequilibre(T):
>>>             cptdes += 1
>>>     print( 'pour N = \%i, on a les proportions'\%N)
>>>     print(cpteq*100/nb, cptdes*100/nb)
```

La fréquence théorique d'appartition d'un tableau binaire équilibré de taille $N \times N$ est $\dfrac{\binom{N^2}{\frac{1}{2}N^2}}{2^{N^2}}$

Dans le tableau suivant, on entre les résultats obtenus sur plusieurs essais avec des échantillons de 1000 à 30000 pour les fréquences d'apparition d'un tableau équilibré

| N = | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **théorie** | **19,6%** | **13,2%** | **9,9%** | **7,96%** | **6,64%** | **5,69%** | **4,98%** | **4,43%** | **3,99%** | **3,63%** | **3,32%** | **3,07%** |
| test 1 | 19,3% | 13,7% | 9,8% | 7,5% | 6,6% | 6% | 4,9% | 4,1% | 3,6% | 3,8% | 3% | 3,1% |
| test 2 | 19% | 13,2% | 10,3% | 8% | 6,4% | 5,6% | 4,6% | 4,7% | 4,1% | 3,7% | 3,6% | 3,2% |
| test 3 | 19,6% | 12,6% | 10,2% | 7,4% | 6,3% | 5,5% | 5,1% | 4,5% | 4,3% | 3,5% | 3,4% | 3% |
| test 4 | 19,7% | 12,1% | 9,6% | 7,9% | 6,6% | 5,6% | 5% | 4,5% | 4% | 3,7% | 3,4% | 3% |
| test 5 | 19,4% | 13,6% | 9,7% | 7,8% | 6,2% | 5,6% | 4,8% | 4,3% | 4,1% | 3,4% | 3,6% | 3,1% |
| test 6 | 19,4% | 12,7% | 10% | 7,8% | 7,5% | 4,9% | 5,4% | 4,6% | 3,8% | 3,4% | 3,2% | 3,1% |
| test 7 | 20,1% | 13% | 9,3% | 8,4% | 7,2% | 5% | 5% | 4% | 3,5% | 3,7% | 3,5% | 3,5% |

Dans le tableau suivant, on entre les résultats obtenus sur plusieurs essais avec des échantillons de 1000 à 30000 pour les fréquences d'apparition d'un tableau totalement déséquilibré

| N = | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **moyenne** | **51,5%** | **45%** | **42%** | **38,5%** | **36%** | **34,5%** | **33%** | **32%** | **31,5%** | **29,8%** | **29,3%** | **28,6%** |
| test 1 | 51,6% | 45,2% | 42,8% | 39,5% | 36,5% | 33,8% | 33,5% | 32,5% | 31,9% | 30,1% | 29,3% | 28,9% |
| test 2 | 51,6% | 44,9% | 41,3% | 38,2% | 37,4% | 34,3% | 32,5% | 31,6% | 31,5% | 29,3% | 29,5% | 28,9% |
| test 3 | 51,4% | 45,1% | 42,2% | 38,4% | 36,1% | 34,3% | 33,6% | 31,6% | 31,2% | 29,8% | 28,8% | 28,2% |
| test 4 | 51,1% | 45,1% | 41,1% | 38,5% | 36,3% | 34,8% | 32,9% | 31,8% | 30,9% | 30,3% | 29,1% | 28,5% |
| test 5 | 51,6% | 44,9% | 42,4% | 38,9% | 36,5% | 34,9% | 33% | 32,5% | 31,2% | 30,2% | 28,6% | 28,8% |
| test 6 | 51,9% | 44,7% | 41% | 38,9% | 35,2% | 35,1% | 32,9% | 31,1% | 31,4% | 29,4% | 29,2% | 28,3% |
| test 7 | 51,8% | 46% | 42,6% | 38,1% | 35,4% | 33,4% | 33,4% | 31,7% | 31,6% | 28,9% | 30,7% | 28,4% |

# 4 Problème : Factorielle et coefficients binomiaux

## 4.1 Partie I : Factorielle

On considère l'algorithme suivant

```
>>> f = 1
>>> for i in range(1, n+1):    # n étant un entier naturel donné
>>>         f = f * i
```

1. État des variables $i$ et $f$ pour l'entrée $n = 6$.

| instruction | f en début de boucle | i | f à la fin |
|---|---|---|---|
| f = 1 | 1 | | 1 |
| boucle | 1 | | 1 |
| i = 1 | 1 | 1 | 1 |
| i = 2 | 1 | 2 | 2 |
| i = 3 | 2 | 3 | 6 |
| i = 4 | 6 | 4 | 24 |
| i = 5 | 24 | 5 | 120 |
| i = 6 | 120 | 6 | 720 |
| sortie de boucle | 720 | 6 | 720 |

2. Soit l'expression mathématique : $v_i = \dfrac{f}{i!}$ où $i$ et $f$ sont les variables de l'algorithme à la fin de chaque itération. Montrons que, tout au long de la structure itérative, n a : $v_i = 1$. Tout d'abord en arrivant dans la boucle, on a $f = 1$ et $i = 1$ donc $v_i = v_1 = 1$. Si on a $v_i = 1$ alors, à l'étape suivante (donc $i + 1$), le '$i$!' est mulitplié par $i + 1$ tout comme le $f$ donc $v_{i+1} = v_i = 1$. Donc $\dfrac{f}{i!}$ **est un invariant de boucle**. En sortant de la boucle, on a $i = n$ et donc $f = n!$

3. On a autant de multiplications que d'itérations donc il y a $n$ **multiplications**

4.

```
>>> def factorielle(n):
>>>     if n ==0: return(1)
>>>     f = 1
>>>     for k in range(1,n+1):
>>>         f = f*k
>>>     return(f)
```

## 4.2 Partie II : Calcul d'un coefficient binomial

1. Voici une fonction **binomial1(n,k)** pour le calcul des coefficients binomiaux :

```
>>> def binomial1(n, k):
>>>         return (factorielle(n) // (factorielle(k) * factorielle(n-k))
```

On a $n$ multiplications pour calculer factorielle(n), $k$ multiplications pour calculer factorielle(k) et $n-k$ multiplications pour calculer factorielle(n-k). Comme il y a en plus une multiplication de factorielles, l'appel de la fonction **binomial1(n,k)** nécessite $2n + 1$ **multiplications d'entiers soit** $O(n)$.

2. (a) Question mathématique Le dénominateur de

$\dfrac{\prod_{i=0}^{k-1}(n-i)}{\prod_{i=0}^{k-1}(i+1)}$ vaut $k!$ et son numérateur est le quotient de $n!$ par $(n-k)!$. Donc on a bien $\binom{n}{k} = \dfrac{\prod_{i=0}^{k-1}(n-i)}{\prod_{i=0}^{k-1}(i+1)}$. Par ailleurs la relation $\binom{n}{k} = \binom{n}{n-k}$ s'obtient aisément par calcul ou raisonnement ensembliste.

(b)

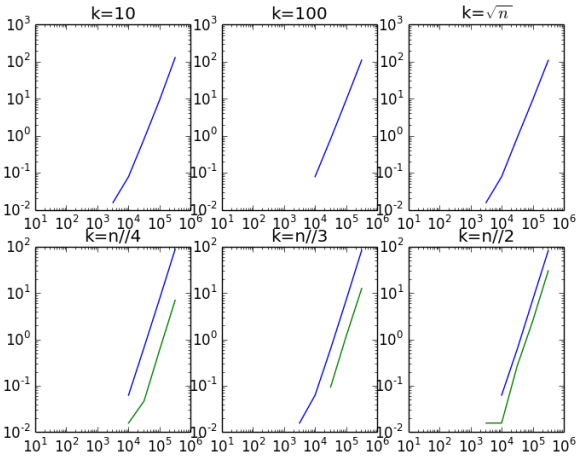
```
>>> def binomial2(n,k) :
>>>     num, denom = 1, 1
>>>     for i in range(0,k) :
>>>         num = num * (n-i)
>>>         denom = denom * (i+1)
>>>     return (num//denom)
```

On a deux multiplications par itérations donc il y a **$2k$ multiplications**

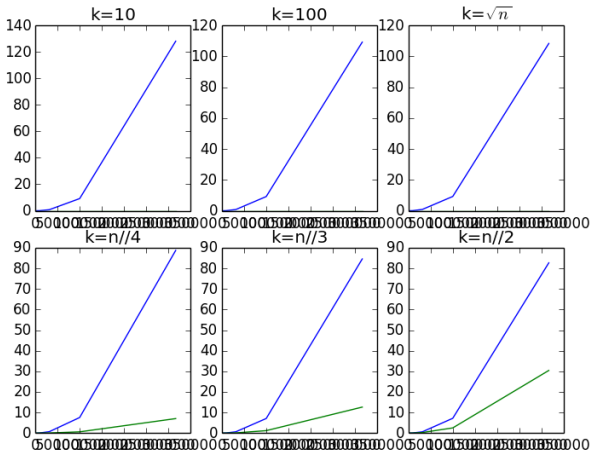
(c) Tableau avec $n = 100000$ puis $n = 316227 = 10^{5.5}$. Les données sont en secondes

| fonction | $k=10$ | $k=100$ | $k=\sqrt{n}$ | $k=n//4$ | $k=n//3$ | $k=n//2$ |
|---|---|---|---|---|---|---|
| binomial1 | 9,2 | 9,2 | 9,26 | 7,55 | 7,07 | 7,17 |
| binomial2 | 0,0 | 0,0 | 0,0 | 0,59 | 1,17 | 2,5 |
| binomial1 | 128,1 | 109,2 | 108,4 | 88,7 | 84,6 | 82,7 |
| binomial2 | 0,0 | 0,0 | 0,0 | 7,03 | 12,61 | 30,42 |

Echelle Logarithmique




Echelle linéaire




3. Remarquant que $\binom{n}{k} = \prod_{i=0}^{k-1}\left(\dfrac{n-i}{i+1}\right)$, un élève définit alors la fonction **binomial3(n,k)** suivante :

```
>>> def binomial3(n, k):
>>>         prod = 1
>>>         for i in range(0,k):
>>>                 prod = prod * (n-i)/(i+1)
```

```
>>>         return (int(prod))
```

Il obtient alors les résultats contradictoires :

```
>>> binomial2(59, 22)
8964377427999630
```

```
>>> binomial3(59, 22)
8964377427999631
```

(a) Question mathématique Si $n$ entier avec $1 \leqslant n \leqslant 124$, on a $\left\lfloor \frac{n}{5} \right\rfloor$ multiples de 5 entre 1 et $n$, dont $\left\lfloor \frac{n}{25} \right\rfloor$ multiples de 25 et aucun multiple de $5^3$ Donc

**la valuation 5-adique de $n!$ est égal à : $\left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{25} \right\rfloor$**

(b) La valuation 5-adique de 59! est 13, celle de 22! est 4 et celle de 37! est 8. Ainsi,

**5 divise $\binom{59}{22} = \frac{59!}{22! \times 37!}$**. En particulier le résultat de binomial3(59, 22) est faux.

L'erreur provient du fait que la division $(n-i)/(i+1)$ est une division flottante. Aussi tous les calculs sont menés dans l'ensemble des flottants pour lequel on n'a que des calculs approchés.

## 4.3 Partie III : Calcul de tous les coefficients binomiaux

De nombreux problèmes nécessitent d'avoir accès à tous les coefficients binomiaux (ou au moins à ceux d'une ligne du triangle de Pascal).

(a) Soit $N \in \mathbb{N}^*$. En utilisant directement **binomial2** pour calculer tous les coefficients binomiaux $\binom{N}{k}$ pour $k$ allant de 0 à $N$, on effectue $2\times1+2\times2+\cdots+2\times N = N\times(N+1)$ multiplications :

**la complexité par cette méthode est $O(N^2)$** En remarquant qu'il suffit de calculer les $\binom{N}{k}$ pour $k$ allant de 0 à $N//2$, on trouverait un nombre de multiplications divisé par 4... mais cela ne change pas la complexité en $O(N^2)$ .

(b)

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

(c) Pour l'obtention de tous les éléments d'une ligne, à partir de la seconde, on effectue $N$ additions. Donc au total, on effectue $N^2$ additions. **La complexité de cette méthode est $O(N^2)$**

On pourrait croire qu'il s'agit de la même complexité que l'algorithme précédent ... sauf qu'ici, ce sont des additions (et non des multiplications) et qu'auparavant on n'avait qu'une ligne : si on voulait tout le triangle de Pascal jusqu'à la ligne $N$, on aurait une complexité de $O(N^3)$ .

(d)

```
>>> def affichepoint(T):
>>>     tab =''
>>>     for ligne in T:
>>>         for coeff in ligne :
>>>             if coeff %2 ==0:
```

```
>>>                 tab = tab + ' '
>>>             else : tab = tab + '*'
>>>         tab = tab + '\n'
>>>     print(tab)
```