I Binôme et plus

1. Écrire une fonction `C(n:int,p:int)->int` qui renvoie la valeur du coefficient binomial $\binom{n}{p}$ en utilisant une programmation itérative (calcul de bas en haut)
2. On définit les coefficients trinômiaux (qui permettent entre autres de développer $(a+b+c)^n$) de la façon suivante : si $i+j+k=n$

$$\binom{n}{i,j,k} = \begin{cases} 1 & \text{si 2 des entiers } i,j,k \text{ sont nuls} \\ \binom{n-1}{i-1,j,k} + \binom{n-1}{i,j-1,k} + \binom{n-1}{i,j,k-1} & \text{sinon} \end{cases}$$

   a) Écrire une fonction `T(i:int,j:int,k:int)->int` récursive « naïve » qui renvoie $\binom{n}{i,j,k}$, où $n=i+j+k$

   b) En déduire deux fonctions récursives mémoïsées à partir de `T` : une en utilisant deux fonctions couplées, la seconde utilisant une sous-fonction.

   c) Écrire une fonction itérative qui renvoie aussi cette valeur.

II Suite de Syracuse

La suite de Syracuse est définie de la façon suivante :

$$u_0 \in \mathbb{N}^* \text{ et } \forall n \in \mathbb{N}, u_{n+1} = \begin{cases} \dfrac{1}{2}u_n & \text{si } n \text{ est pair} \\ 3u_n+1 & \text{si } n \text{ est impair} \end{cases}$$

On conjecture que, quelle que soit la valeur de u_0, il existe un indice n pour lequel $u_n = 1$; la suite devient alors périodique et prend alternativement les valeurs $1, 4, 2, 1, 4, 2, \ldots$.
On appelle temps de vol de l'entier k, que l'on va note $tv(k)$, le premier entier n pour lequel la suite de Syracuse initialisée avec $u_0 = k$ prend la valeur 1.

1. Écrire une fonction `suivant(k:int)->int` qui renvoie le terme u_{n+1} si $u_n = k$.
2. En remarquant que $tv(u_n) = 1 + tv(u_{n+1})$, écrire une fonction récursive `tv(k:int)->int` qui renvoie le temps de vol de $k \in \mathbb{N}^*$.
3. On souhaite maintenant écrire une fonction `tvMax(n:int)->int` qui renvoie le maximum des temps de vols des entiers de $[\![1, n]\!]$. Coder cette fonction ; afin de ne pas recalculer plusieurs fois les mêmes valeurs, on utilisera un dictionnaire pour mémoriser les calculs déjà faits.
4. Déterminer le temps de vol de l'entier 3 et expliquer pourquoi une programmation de bas en haut, sans utiliser un dictionnaire serait difficile.

III Optimiser un produit matriciel

Le produit matriciel est associatif mais les deux façons de calculer ABC, selon le premier produit effectué, ne conduisent pas à effectuer le même nombre de multiplications entre scalaires (si les matrices ne sont pas carrées). Plus précisément, si $A \in \mathcal{M}_{n_1,n_2}(\mathbb{R})$, $B \in \mathcal{M}_{n_2,n_3}(\mathbb{R})$ et $C \in \mathcal{M}_{n_3,n_4}(\mathbb{R})$ alors le produit $(AB)C$ nécessite $n_1n_2n_3 + n_1n_3n_4$ produits de scalaires alors que $A(BC)$ en nécessite $n_2n_3n_4 + n_1n_2n_4$.

1. Nombre de parenthésages : si on note c_n le nombre de façons d'effectuer le produit de n matrices (donc de placer les parenthèses de façon à découper le calcul avec des produits 2 par 2) alors on a

$$c_{n+1} = \sum_{k=1}^{n} c_k c_{n+1-k}$$

   En déduire une fonction `c(n:int)->int` récursive « efficace » qui renvoie le nombre de parenthésages d'un produit de n matrice (supposé possible).

2. On note $N(n_1, \ldots, n_{k+1})$ le nombre minimal de produits de scalaires à effectuer pour calculer le produit matriciel $A_1 \times \cdots \times A_k$, avec $A_i \in \mathcal{M}_{n_i,n_{i+1}}(\mathbb{R})$. Justifier que

$$N(n_1, \ldots, n_{k+1}) = \min_{i \in [\![1,k-1]\!]} \left\{ N(n_1, \ldots, n_{i+1}) + N(n_{i+1}, \ldots, n_{k+1}) + n_1 n_{i+1} n_{k+1} \right\}$$

a) Écrire une fonction récursive « naïve » `N(t:tuple)->int` qui prend en argument le tuple $(n_1, \dots, n_{k+1})$ et qui renvoie le nombre de multiplication minimal à effectuer. Vérifier que sa complexité est exponentielle.

b) Mémoïser cette fonction avec un dictionnaire.

c) Écrire une fonction itérative efficace renvoyant le même résultat ; on pourra introduire un tableau `T` (liste de listes) pour lequel `T[i][j]` contient le nombre de produits à effectuer pour calculer $A_i \times \cdots \times A_j$. Quelle est la complexité de cette fonction ?

d) Écrire une fonction `ordre(t:tuple)->list` qui prend en argument le tuple $(n_1, \dots, n_{k+1})$ et qui renvoie la liste des numéros des produits à effectuer dans l'ordre pour minimiser le coût des calculs.