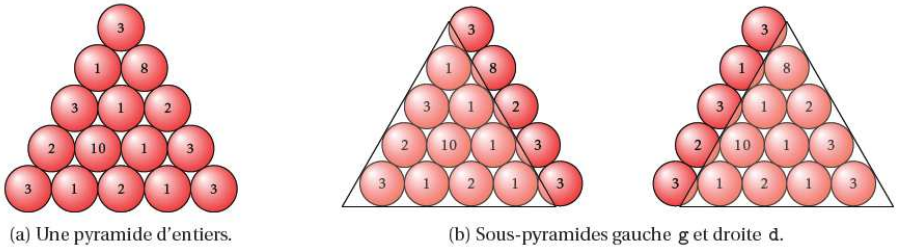
TD ITC2^2 N° 7: PROGRAMMATION DYNAMIQUE: EXEMPLES CLASSIQUES

EXERCICE N°1: Recherche du chemin optimal dans une pyramide de nombres

On considère une pyramide de nombres entiers positifs, de hauteur n. On souhaite dégager une méthode de détermination de la somme maximale obtenue en parcourant la pyramide de haut en bas (TOP-BOTTOM); c'est le critère d'optimisation! A chaque passage d'un niveau i à un niveau $i+1$ inférieur, seuls les deux éléments de niveau $i+1$ situés immédiatement en dessous du nombre considéré au niveau i sont pris en considération pour le calcul de la somme. La figure 1a représente une pyramide de hauteur 5 dont la somme maximale est 24. La somme est obtenue en passant successivement par les nombres 3, 8, 1, 10, 2.



(a) Une pyramide d'entiers.

(b) Sous-pyramides gauche `g` et droite `d`.

Figure 1: Pyramide de nombres

Une pyramide est représentée par un tableau à deux dimensions t de taille $n \times n$. On note $t_{i,j}$ le nombre situé à la ligne i et à la colonne j . La pyramide de la figure 1*a* est représentée par le tableau suivant.

$$\begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 1 & 8 & 0 & 0 & 0 \\ 3 & 1 & 2 & 0 & 0 \\ 2 & 10 & 1 & 3 & 0 \\ 3 & 1 & 2 & 1 & 3 \end{bmatrix}$$

❶ Écrire une fonction `max_nb(x,y)` qui renvoie le plus grand des nombres x et y.

❷ Ecrire une fonction `gen_tab(n)` permettant de définir un tableau t **associé à une pyramide de hauteur** n **fixée** dont les éléments sont tirés au hasard entre 1 et 10 inclus. On pourra utiliser la commande `randint(a,b)` du module `random` qui génère un entier aléatoire compris entre a et b inclus.

❸ On se propose de calculer cette somme par une méthode récursive. La pyramide de hauteur n est décomposée en deux sous-pyramides g et d (figure 1b) de hauteur $(n-1)$ chacune. La somme maximale est calculée en additionnant l'élément $t_{1,1}$ et la plus grande somme obtenue à partir des sous-pyramides g et d. Cette procédure est de nouveau appliquée aux sous-pyramides g et d. Elle s'arrête dès que la base de la pyramide est atteinte (condition de terminaison). On désigne par $s_{i,j}$ la somme maximale pour une sous-pyramide de sommet l'élément $t_{i,j}$.

- **a**· Quelle est la condition de terminaison de la somme précédemment définie?
- **b**· Etablir une relation de récurrence entre les sommes $s_{i,j}$.
- **c**· Ecrire une fonction `s_rec(t,i,j)` qui calcule récursivement $s_{i,j}$.
- **d**· Comment calculer la somme maximale associée à la pyramide complète?
- **e**· Déterminer la complexité de cet algorithme en terme d'appels récursifs. Commenter cette complexité.

❹ On se propose de reprendre le problème par une méthode itérative assurant une complexité temporelle bien plus faible.

Cette méthode qui examine la pyramide de bas en haut (type "BOTTOM-UP") consiste à remplacer chaque élément d'une ligne i à la colonne j par la somme maximale qu'il peut former avec l'un de ses deux éléments directement **adjacents** de la ligne inférieure $i+1$. Ainsi, en itérant cette méthode, **jusqu'au sommet de la pyramide**, ce dernier prendra finalement la valeur de la somme maximale recherchée.

- **a**· Proposer une fonction `s_iter(t)` qui applique cette méthode itérative au tableau t.
- **b**· Estimer le nombre d'opérations (répétitions, conditions) nécessaires au calcul de la somme par cette méthode en fonction de n. Commenter, et comparer en particulier la performance de cet algorithme itératif dynamique à celle de la méthode récursive employée plus haut.
- **c**· A partir de la matrice des sommes cumulées obtenue par la méthode dynamique, proposer une stratégie de reconstitution du chemin emprunté dans la pyramide pour obtenir la somme maximale. Proposer une fonction python

`recons_chemin(t:array)` → `str` qui renvoie ce chemin sous la forme d'une chaine de caractère constituée de **"B"** (Bas) et **"D"** (Diagonal), avec "B" pour un déplacement vers le terme matriciel juste en dessous, et "D" un déplacement vers le terme matriciel en dessous et à droite.

EXERCICE N°2: **Partition équilibrée d'un tableau d'entiers positifs**

On considère une liste L d'entiers positifs.

On veut établir un algorithme qui partitionne cette liste en deux sous-listes L_1 et L_2 telles que les sommes des entiers des sous-listes soient les plus proches possibles l'une de l'autre. En d'autres termes, on veut dégager L_1 et L_2 telles que:

- $L_1 \cup L_2 = L$
- $L_1 \cap L_2 = \emptyset$
- $|S(L_1) - S(L_2)|$ minimale avec $\quad S(J) = \sum_{x \in J} x \quad$ pour $J \subset L$

❶ Notons $S = S(L)$. Montrer que rechercher $|S(L_1) - S(L_2)|$ minimale revient à rechercher $\left|\frac{S}{2} - S(L_1)\right|$ minimale.

❷ Quelle est en fonction de n, cardinal de L, la complexité d'une recherche de L_1 par force brute?
De manière plus formelle, on cherche donc ici à déterminer la liste L_1 telle que:

$$\left|\frac{S}{2} - \sum_{a \in L_1} a\right| = min\left(\left\{\left|\frac{S}{2} - \sum_{x \in X} x\right| / X \subset L\right\}\right)$$

❸ On propose dans un premier temps d'aborder ce problème par la récursivité selon le schéma suivant:

- soit $L[0]$ fait partie de la partition minimale et alors on doit rechercher ensuite la sous-liste $L[1:]$ qui se rapproche le plus de $\frac{s}{2} - L[0]$
- soit $L[0]$ n'en fait pas partie et alors on doit rechercher ensuite la sous-liste $L[1:]$ qui se rapproche le plus de $\frac{s}{2}$

a· Proposer une fonction récursive `part(L:list,S12:int)` → `list` qui prend en argument la liste L et la demi-somme $S12 = \frac{\sum_{x \in L} x}{2}$, et renvoie la liste L_1 correspondant à une partition équilibrée.

b· Calculer la complexité de cet algorithme en terme de récursions, bâtir l'arbre des appels récursifs, et conclure sur l'efficacité de cette méthode.

c· Proposer une amélioration de la méthode récursive en faisant appel à la mémoïsation.

❹ APPROCHE PAR LA PROGRAMMATION DYNAMIQUE

On peut contourner cet écueil de la méthode récursive en procédant par une méthode de programmation dynamique de type "BOTTUM-UP.

Le principe est de calculer toutes les sommes que l'on obtient avec les listes partielles ne comportant que les k premiers termes: en ajoutant le $k+1^{\text{ième}}$ terme, les sommes obtenues sont les mêmes que celles obtenues au rang précédent auxquelles on ajoute celles obtenues lorsqu'on ajoute le $k+1^{\text{ième}}$ terme (propriété (e)).

On peut facilement présenter cette méthode sous forme d'un tableau T (de booléens) de taille $[len(L)+1, S+1]$ dans lequel pour $0 \leq i \leq len(L)$ et $0 \leq j \leq S$ T_{ij} est `True` si la somme j est atteignable avec des entiers pris dans l'ensemble $L[:i]$, et `False` sinon.

Par exemple, avec la liste $[2,4,5,3]$, on a $S = 14$ et le tableau correspondant est:

$j\rightarrow$ / $i\downarrow$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	T	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	T	F	T	F	T	F	F	F	F	F	F	F	F
3	T	F	T	F	T	T	T	T	F	T	F	T	F	F	F
4	T	F	T	T	T	T	T	T	T	T	T	T	T	F	T

A la lecture de ce tableau, on peut remarquer les propriétés suivantes:

- Pour $i = 0$, on a $T[0,j] = False$ sauf pour $j = 0$.
- Pour $i > 0$, si $T[i,j] = True$ alors il existe un sous-ensemble de $L[:i]$ dont la somme des éléments vaut j (en particulier on a $T[i>0,0] = True$) et on a:
  - soit $T[i-1,j] = True$
  - soit $T[i-1, j-L[i-1] = True$

Ecrire une fonction python `tableau_somme(L:list)` → array prenant en argument la liste d'entiers L et renvoyant le tableau correspondant de booléens.

❺ Le tableau établi plus haut va permettre de reconstituer la liste L_1 recherchée dont la somme des éléments est la plus proche possible de $\lfloor\frac{S}{2}\rfloor$.
On commence par rechercher sur la dernière ligne, c'est à dire **en prenant en compte**

tous les entiers de la liste L, la plus grande valeur j_m de j inférieure ou égale à $\frac{S}{2}$ telle que $T[len(L), j_m] = True$.

a· Déterminer la valeur de j_m pour la liste $[2,4,5,3]$.

b· Par définition de j_m, si on trouve un indice i tel que $T[i+1, j_m] = True$ et $T[i, j_m] = False$, alors cela signifie que la somme j_m est atteignable avec la sous-liste $L[:i+1]$ mais pas avec $L[:i]$.

Quelle somme peut-on obtenir avec la sous-liste $L[:i]$?

c· Déduire de la question précédente une stratégie pour reconstituer la liste L_1.

Ecrire alors la fonction `Partition_dyn(L:list) → list` recevant la liste L et renvoyant la liste L_1 à partir de cette stratégie.