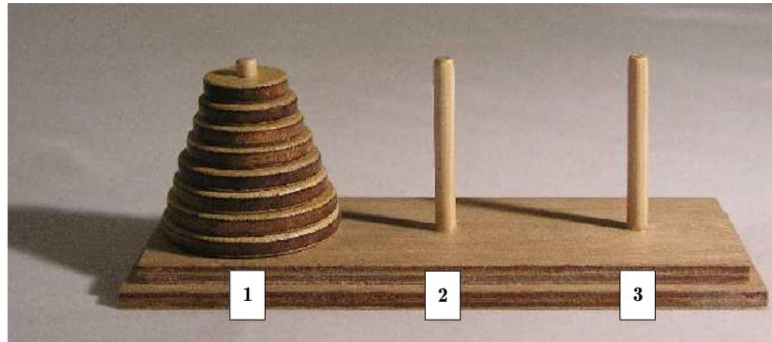
TP5 – PROGRAMMATION DYNAMIQUE

1. TOUR DE HANOÏ

Les Tours de Hanoï est un jeu inventé par le mathématicien Édouard Lucas en 1883. Il est constitué de trois piquets verticaux, notés 1, 2 et 3 et de ***n disques*** *superposés de tailles strictement décroissantes avec un trou au centre et enfilés autour du piquet 1.*

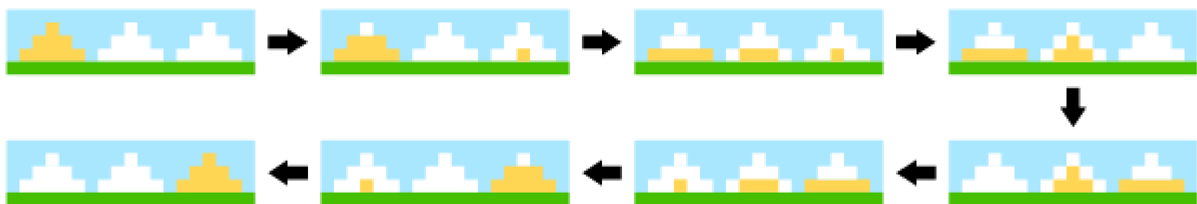


Le but du jeu consiste à déplacer l'ensemble des disques pour que ceux-ci se retrouvent enfilés autour du piquet 3 en respectant les règles suivantes :

- *les disques sont déplacés un par un ;*
- *un disque ne doit pas se retrouver au-dessus d'un disque plus petit. (On suppose évidemment que cette dernière règle est également respectée dans la configuration de départ).*

Ce problème se résout facilement de manière récursive. Pour déplacer ***nb*** *disques de la position* ***pos_init*** *à la position* ***pos_fin*** *il faut :*

- *déplacer les* ***nb-1*** *premiers éléments vers la position restante* ***pos_inter****,*
- *déplacer l'élément restant (le plus gros) de la position* ***pos_init*** *à la position* ***pos_fin****,*
- *déplacer la pile des* ***nb-1*** *éléments de la position* ***pos_inter*** *à la position* ***pos_fin*** *sur le plus gros déjà en place.*



Q1. Écrire sous Python une procédure (`print` au lieu de `return`) récursive ***hanoi****(nb, pos_init, pos_fin) résolvant le problème en affichant les étapes sous la forme « Déplacer le disque de 1 vers 0 ».*

« Interface graphique »

Ouvrir le programme « ***hanoi_graph_joli.py*** *» permettant de visualiser les coups.*

Q2. Lire et comprendre le programme complet, notamment comment est calculé le nombre de coups.
Q3. Améliorer le programme en demandant à l'utilisateur de rentrer les paramètres de la fonction de Hanoi et qu'elle s'applique automatiquement (rappel : on utilisera ***input****)*
Q4. Mise en place de balises temporelles du module time

A l'aide du module time, mettre en place un dispositif permettant d'estimer le temps utilisé par la fonction globale, en fonction du nombre de disques.

Q5. Tester le programme avec n=5, 6, 7, 8, 9, 10.

Q6. Conjecturer la complexité temporelle (nb de coups) et la démontrer.
Q7. Peut-on utiliser les principes de la programmation dynamique pour réduire cette complexité ?

2. ETUDE D'UNE ENTREPRISE

Une entreprise de livraison dispose de plusieurs locaux en France et chacun possède plusieurs camions de livraisons. Celle-ci souhaite optimiser le chargement de ses camions pour diminuer ses frais de fonctionnement.

Partie I - Optimisation du chargement

Chaque camion de l'entreprise peut charger une cargaison jusqu'à un poids maximal noté P_{max}. L'entreprise dispose de différentes informations provenant de ses clients :

- le poids de chaque produit p_i (chaque client propose un seul produit) ;
- la valeur v_i associée au transport de chaque produit : c'est-à-dire l'argent gagné par l'entreprise si elle réalise le transport de ce produit.

En considérant que l'entreprise dispose de n clients, l'entreprise cherche donc à trouver une liste d'indices notée I contenue dans $\{1, \ldots, n\}$ telle que :

$$\sum_{i \in I} p_i \leq P_{max} \quad \text{(respect du poids maximal)} \qquad (1)$$

et

$$\sum_{i \in I} v_i \text{ soit maximal} \quad \text{(optimation du profit pour l'entreprise).} \qquad (2)$$

Dans toute la suite, les poids seront donnés en centaines de kilogrammes et les valeurs en centaines d'euros.

Nous introduisons une méthode récursive pour résoudre le problème d'optimisation :

- pour chacun des produits, deux choix sont possibles : il fait partie de la cargaison ou non ;
- la récursivité s'effectuera sur la liste des indices de `Pr` : le premier appel de la fonction se fera en utilisant l'indice n, puis l'indice $n-1$ et ainsi de suite jusqu'à l'indice 0 (correspondant au cas où il n'y a plus de produits) ;
- pour $i \in \{0, 1, \ldots, n\}$ et $\omega \in \{0, 1, \ldots, P_{max}\}$, on note $S(i, \omega)$ la valeur maximale cumulée des produits que l'on peut placer dans un camion d'une capacité maximale (en poids) de ω avec la liste constituée des i premiers produits de `Pr`.

On pose alors la relation de récursivité suivante :

$$S(i,\omega) = \begin{cases} 0 & \text{si } i = 0 \\ S(i-1,\omega) & \text{si } i > 0 \text{ et } p_i > \omega \\ \max(S(i-1,\omega), v_i + S(i-1, \omega - p_i)) & \text{si } i > 0 \text{ et } p_i \leq \omega. \end{cases} \qquad (3)$$

Q8. Justifier la terminaison de l'algorithme associé à la relation de récursivité précédente.
Q9. En vous basant sur la relation (3), écrire une fonction `def recur(P,V,i,w)` *ayant pour argument, les listes de poids* `P`*, de valeurs* `V`*, un indice* `i` *un poids* `w` *et renvoyant* $S(i,\omega)$
Q10. Tester votre fonction avec `Pr=[1,2,3,4] ; P=[3,2,1,4], V=[4,3,1,9] et Pmax=8` *et vous obtiendrez 14.*

On souhaite améliorer la méthode récursive. Nous allons procéder en mémorisant des calculs déjà effectués. Voici le principe : nous allons stocker les valeurs $S(i,\omega)$ dans un tableau `Memoire`, de taille `(n+1)×(Pmax+1)`, initialisé au départ avec des éléments tous égaux à -1.

Si la valeur de $S(i,\omega)$ a déjà été calculée, l'élément d'indice (i,ω) de ce tableau `Memoire` ne sera plus égal à -1 : on renverra donc directement la valeur. Sinon, on la calculera en suivant le principe précédent et on la stockera dans le tableau avant de la renvoyer.

Nous faisons le choix de représenter les tableaux comme des listes de listes.

Q11. Donner l'instruction permettant de créer le tableau `Memoire` initialisé avec des coefficients égaux à -1, en supposant Pmax et n connus. Quel serait l'intérêt d'utiliser des `array` ?

Q12. Ecrire une fonction `recur2` en suivant le principe expliqué et permettant d'améliorer la fonction `recur`. La variable `Memoire` sera utilisée comme une variable globale.

Avec les données suivantes :

- `P = [5,3,3,3];`
- `V = [4,3,1,1];`
- `Pmax = 8;`
- `Memoire un tableau de taille 5 × 9;`

et en exécutant `recur2(P,V,len(P),Pmax,Memoire)`, on obtient alors la valeur 7.

Q13. Compter le nombre de coup dans les 2 cas (recur et recur2) et utiliser le module time pour voir la complexité temporelle pour des listes P et V de 40 valeurs aléatoires (à implémenter en utilisant la fonction `randint(1,100)` du module `random`) et Pmax=200. Comparer les 2.

2. PROBLEME DU SAC A DOS (GENERALISATION)

Je vous propose ici de généraliser la méthode. Comme vous l'avez vu précédemment, la récursivité, même en programmation dynamique est extrêmement couteuse. Nous allons donc voir une méthode itérative qui utilise la programmation dynamique.

Soit E un ensemble de n éléments à ranger dans un sac à dos de capacité maximale W (son volume);

- chaque élément e a une valeur v (on les numérote de 1 à n)
- chaque élément a un poids w (on les numérote de 1 à n)

On dispose d'un prédicat c sur les sous-ensembles de E.

On cherche à trouver un sous-ensemble $F \subseteq E$ tel que:

- le prédicat $c(F)$ est évalué à Vrai (souvent on demande que la somme des poids des éléments dans F ne dépasse pas la capacité W
- la somme des valeurs des éléments de F est maximale ($\sum_{e\in F} v(e)$).

La programmation dynamique permet de scinder les entrées du problème en autant de sous-ensemble que nécessaire. Le problème est résolu pour chacun des sous-ensembles en utilisant les solutions précédentes pour calculer la valeur du sous-ensemble courant (comme dans Fibonacci). Il faut cependant qu'il n'y ait qu'un nombre polynomial de sous-problèmes.

Remarquons que la recherche naïve d'une solution obligerait de parcourir les 2^n sous-ensembles possibles de E, donnant lieu à un algorithme clairement exponentiel !

Résolution :

On se donne un sac à dos de capacité 7 et un ensemble de 3 éléments ordonnés selon le rapport v/w. Les poids et les valeurs sont décrits sous la forme d'un dictionnaire pour que la numérotation commence à 1 et non à 0.

```
w={1:5,2:4,3:3}
v={1:10,2:7,3:5}
```

On définit une relation de récurrence $Opt(i, j)$ où $Opt(i, j)$ représente la valeur optimale de poids maximal pour un sous-ensemble qui utilise des éléments de $\{1, \ldots, i-1\}$ de poids maximal j:

$$Opt(i,j) = \begin{cases} 0 & \text{si } i = 0 \\ Opt(i-1,j) & \text{si } w_i > j \text{ pour } 1 \leq i \leq n \text{ et } 0 \leq j \leq W \\ \max\{Opt(i-1,j), v_i + Opt(i-1,j-w_i)\} & \text{sinon} \end{cases}$$

La relation de récurrence signifie :

- **cas 1 :** l'élément i n'est pas ajouté. Dans ce cas, on conserve la valeur maximale obtenue en ayant déjà considéré les éléments $\{1, \ldots, i-1\}$ qui a été précédemment calculée en utilisant le poids maximal j : $Opt(i-1, j)$
- **cas 2 :** l'élément i est ajouté. Il est de poids w_i et de valeur v_i. Pour cela, il faut que la capacité restante $j \geq w_i$
  - on met à jour la capacité restante qui devient $j - w_i$
  - on met a jour la valeur optimale qui prend le maximum entre:
    - la valeur précédemment calculée (i.e. on n'a pas retenu l'élément i) qui était meilleure
    - on ajoute la valeur de l'élément i, v_i et on calcule la nouvelle valeur optimale de $Opt(i-1, j-w_i)$

Plusieurs méthodes permettent de calculer cette récurrence, comme pour Fibonacci: la résolution directe, la résolution directe avec une mémo-fonction ou par un calcul itératif en partant de la base.

Illustrons cette dernière méthode pour laquelle on définit un tableau V à n lignes et W colonnes qui va mémoriser les valeurs de Opt.

On utilise le paradigme de la programmation dynamique qui va calculer la solution aux sous-problèmes une seule fois et le mémoriser dans une table pour que ce calcul puisse être réutilisé.

Etape 1 : On construit un tableau $V[0 \ldots n, 0, \ldots, W]$. Pour $1 \leq i \leq n$ et $0 \leq j \leq W$, la case $V[i, j]$ va mémoriser la valeur maximale des sous-ensembles de taille au plus i.

Si on arrive à calculer toutes les cases de ce tableau, la case $V[n, W]$ contiendra la valuer maximale des éléments qui peuvent être rangés dans le sac à dos, i.e. la solution à notre problème.

Etape 2 : On construit par récurrence la valeur d'une solution optimale en termes de solutions à des sous-problèmes:

- **initialisation :** On pose:
  - $V[0,j] = 0$ pour $0 \leq j \leq W$ (on n'a pas gardé d'élément)
  - $V[i,j] = -\infty$ pour $j < 0$ (entrée interdite)
- **récursion :** On utilise pour $1 \leq i \leq n, 0 \leq j \leq W$

$$V[i,j] = \max\{V[i-1,j], v_i + V[i-1, j-w_i]\}$$

Etape 3 : On calcule les valeurs de $V[i,j]$ de façon itérative en partant de l'initialisation de V (cf. étape 2) et en mettant à jour

$$V[i,j] = \max\{V[i-1,j], v_i + V[i-1, j-w_i]\}$$

ligne à ligne.

Cet algorithme a une complexité en $O(n.\ W)$ car il doit calculer toutes les case du tableau pour trouver la valeur optimale.

```
from sympy import *
def solsad(v,w,n,weight):
   V=[x[:] for x in [[0]*(weight+1)]*(n+1)]
   for i in range(1,n+1):
      for j in range(weight+1):
         if w[i]<=j: #on ajoute l'élement i
            V[i][j]=max(V[i-1][j],v[i]+V[i-1][j-w[i]])
         else : #l'élément n'est pas ajouté
            V[i][j]=V[i-1][j]
   return V[n][weight]

#TESTER
print(solsad(v,w,3,7))
```

L'algorithme nous retourne la valeur qui maximise le choix du sous-ensemble, obtenu pour les éléments 2 et 3 (on les retrouve à partir des données du problème). Cependant, si on obtient la valeur optimale, l'algorithme ne fournit pas les éléments qui ont été sélectionnés. Pour obtenir les élements qui ont été choisis, il faut ajouter un tableau `memo` qui va mémoriser les élements choisis après avoir été initialisé à 0.

Q14. Améliorer votre fonction en mettant ce tableau `memo` *.*

Le test : `sad(v,w,3,7)`

Donne : poids de `12` avec les éléments `[3,2]` et la table `memo et V.`

$$\left(12, \quad [3, \quad 2], \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10 \\ 0 & 0 & 0 & 0 & 7 & 10 \\ 0 & 0 & 0 & 5 & 7 & 10 \end{bmatrix}\right.$$

Le résultat signifie qu'on a trouvé une solution de poids 12 en sélectionnant les éléments 2 et 3 (on ignore la première ligne de la matrice `memo` qui correspond à l'initialisation à 0). Les éléments sélectionnés vérifient la propriété suivante:

- si `memo`$[n, W]$ est à 1, alors l'élément n est dans le sous-ensemble et on regarde alors pour `memo`$[n-1, W-w_n]$
- si `memo`$[n, W]$ est à 0, alors l'élément n n'est pas dans le sous-ensemble et on regarde alors pour `memo`$[n-1, W]$

Pour notre exemple, il faut regarder successivement les valeurs suivantes de `memo` pour (n, W): $(3, 7)$ (3 est choisi) puis $(2, 7-w_3) = (2, 7-3=4)$ (2 est choisi) puis $(1, 4-w_2) = (1, 4-4=0)$ et on a fini.