TD3 : Programmation dynamique

Exercice 1. *Problème de sac-à-dos (knapsack) en programmation dynamique*

On dispose de n objets ayant des poids $w_1, \dots, w_n$ (positifs) et des valeurs d'utilité $c_1, \dots, c_n$ (positives). On veut choisir des objets parmi les n pour les mettre dans un sac ayant une capacité maximale W en cherchant à maximiser l'utilité totale.

Pour modéliser ce problème, on introduit les variables binaires $x_i = 1$ si on prend l'objet i et $x_i = 0$ sinon. Le problème se s'écrit alors

$$(P) : \qquad \max_{x \in D_n(W)} \Big(F(x) = \sum_{j=1}^{n} c_j x_j \Big)$$

où

$$D_n(W) = \Big\{ (x_1, \dots, x_n) \in \{0,1\}^n \text{ tels que } \sum_{j=1}^{n} w_j x_j \leq W \Big\}$$

On définit la valeur $p_i(w)$ comme la valeur d'utilité maximale en considérant seulement les i premiers objets avec une capacité maximale w.

1. Ecrire le problème $\mathcal{P}_i(w)$ associé à la valeur $p_i(w)$ en fonction des variables $(x_1, \dots, x_i)$.
2. Soit $(x_1, \dots, x_i)$ une solution optimale de $\mathcal{P}_i(w)$. Que dire de la solution si $w_i > w$? Montrer que dans ce cas, $p_i(w) = p_{i-1}(w)$.
3. On suppose que $w_i \leq w$. Montrer que

$$p_i(w) = \max_{x_i \in \{0,1\}} \left(c_i x_i + p_{i-1}(w - w_i x_i) \right) \tag{1}$$

   On montrera l'inégalité dans un sens puis dans l'autre.
4. Ecrire la formule de récurrence complète pour $p_i(w)$. La valeur $p_n(W)$ donne alors le maximum F de (P).
5. *Un exemple.* Résoudre par programmation dynamique le problème de sac-à-dos avec les données

$$w = \begin{pmatrix} 1 \\ 3 \\ 2 \\ 5 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 4 \\ 5 \\ 7 \end{pmatrix}, \quad W = 7$$

6. *Implémentation en python.*
   (a) *Version récursive.* Ecrire un script python pour calculer $p_n(W)$ de manière récursive avec une fonction d'entête

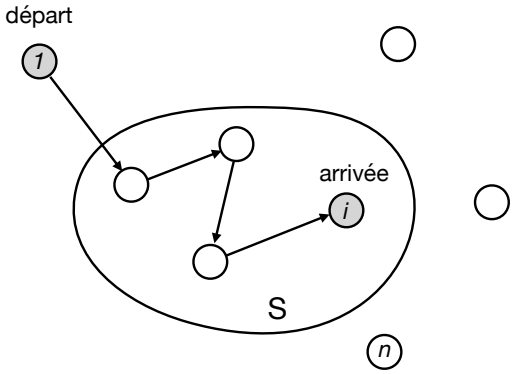
```
def knapsack_recursive(weights, values, w, i):
```

   Il doit y avoir des appels récursifs à la fonction dans la fonction elle-même.
   (b) *Version vectorielle.* On suppose que les poids sont tous *entiers*. A partir de la formule de récurrence établie précédemment, on peut alors calculer un array 2D `p[i,w]` pour $i = 0, \dots, n-1$ et pour $w = 0, \dots, W$. Ecrire le script `python` correspondant.

Exercice 2. *Programmation dynamique pour le TSP*

Dans le problème du voyageur de commerce, il faut passer une et une seule fois par n villes et revenir à la ville de départ en minimisant la distance totale parcourue. La ville de départ est la ville n°1. Pour modéliser le TSP par programmation dynamique, on introduit :

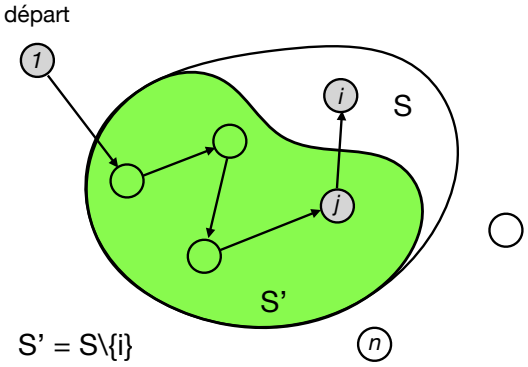
- un sous-ensemble $S \subseteq \{2, \dots, n\}$ de villes ;
- pour un ville $i \in S$, on note $g(S, i)$ la longueur minimale d'un trajet qui :
  - commence en 1,
  - parcourt toutes les villes de S (dans un ordre quelconque),
  - termine en i.



On a alors la formule de récurrence suivante.

- Si $|S| = 1$ i.e. $S = \{i\}$, $i \neq 1$ alors $g(S, i) = d(1, i)$.
- Si $|S| > 1$ alors

$$g(S, i) = \min_{j \in S \setminus \{i\}} \Big(d(j, i) + g(S \setminus \{i\}, j) \Big) \tag{1}$$



La longueur minimale est alors obtenue par :

$$L_{\min} = \min_{i \in [\![2, n]\!]} \Big(d(i, 1) + g(\{2, \dots, n\}, i) \Big) \tag{2}$$

1. On veut établir (2). Montrer que la fonction $g(S, i)$ donne bien le coût minimal pour visiter toutes les villes de S en terminant par i. Procéder par récurrence sur la taille des sous-ensembles S : supposer que pour tout sous-ensemble S de taille k (où $1 \leq k < n$), la fonction $g(S, i)$ donne bien le coût minimal pour visiter toutes les villes de S en terminant par i.

ALGORITHME.

```
# Initialisation : cas |S|=1
Pour i de 2 à n
    g({i}, i) = d(1,i)
fin pour

# Cas |S|>1
Pour j de 2 à n-1
    Pour tous les S de {2,...,n} avec |S|=j
        Pour tout i de S
            g(S,i) = min_{j dans S\{i}} ( d(j,i) + g(S\{i},j) )
        fin pour
    fin pour
fin pour
```

2. Implémenter en python l'algorithme de programmation dynamique. Vous pourrez utiliser la fonction `combinaisons` du module `itertools` qui engendre toutes les combinaisons possibles d'un ensemble donné. Les fonctions g seront définies dans un *dictionnaire* python. Les sous-ensembles S seront des ensembles python *immuables* contenant les villes, pour pouvoir être utilisés comme clefs dans le dictionnaire g. Utiliser pour cela la fonction `frozenset()`.
3. Tester avec la matrice des distances

$$d = \begin{pmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{pmatrix}$$

Vous devez trouver une longueur minimale de 80.