Corrigé du TP de programmation dynamique

Première partie

L'algorithme de Floyd-Warshall

1. On raisonne par récurrence avec l'hypothèse $\mathscr{P}(k)$: « Pour tous $i, j \in [\![0, n-1]\!]$, le nombre $a_{i,j}^{(k)}$ est égal au poids d'un chemin de poids minimal reliant le sommet i au sommet j en ne passant que par des sommets parmi $0, 1, \ldots, k-1$ ».

   Initialisation $\mathscr{P}(0)$ est vraie vu la définition de $A^{(0)} = M$.

   Hérédité Soit $k \in [\![0, n-1]\!]$ tel que $\mathscr{P}(k)$ est vraie, et $i, j \in [\![0, n-1]\!]$.

   Alors

   - s'il existe un chemin de poids minimal de i vers j en ne passant que par des sommets parmi 0, 1, …, k, il y a deux cas possible :
     - ⋆ soit ce chemin ne passe pas par le sommet k, il est donc de poids (minimal) $a_{i,j}^{(k)}$ par hypothèse de récurrence ;
     - ⋆ soit ce chemin ne passe pas par le sommet k et par propriété de sous-optimalité du plus court chemin, les sous-chemins reliant le sommet i à k et k à j sont eux aussi de poids minimal, le poids du chemin est donc $a_{i,k}^{(k)} + a_{k,j}^{(k)}$.
   - S'il n'existe pas de chemin de i vers j en ne passant que par des sommets parmi 0, 1, …, k, alors il n'en existe pas en ne passant que par des sommets parmi 0, 1, …, $k-1$ donc par HR, $a_{i,j}^{(k)} = +\infty$ et soit il n'existe pas de chemin de i vers k ne passant que par des sommets parmi 0, 1, …, $k-1$ soit il n'existe pas de chemin de k vers j ne passant que par des sommets parmi 0, 1, …, $k-1$ donc par HR, $a_{i,k}^{(k)} + a_{k,j}^{(k)} = +\infty$

   Ainsi, on a bien $a_{i,j}^{(k+1)} = \min\left(a_{i,j}^{(k)}, a_{i,k}^{(k)} + a_{k,j}^{(k)}\right)$ poids d'un chemin de poids minimal de i vers j en ne passant que par des sommets parmi 0, 1, …, k, ce qui établit la récurrence.

2. Finalement, pour tous $i, j \in [\![0, n-1]\!]$, $a_{i,j}^{(n)}$ correspond bien au poids d'un chemin de poids minimal de i vers j en ne passant que par des sommets parmi 0, 1, …, k.

3. Ici, pas besoin de recopie de la matrice, vu l'ordre dans lequel les éléments sont modifiés.

   Python

```python
def FloydWarshall(M):
    n = len(M)
    A = [[M[i][j] for j in range(n)] for i in range(n)]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                A[i][j] = min(A[i][j], A[i][k] + A[k][j])
    return A
```

4. On obtient une complexité spatiale en $\boxed{\mathcal{O}(n^2)}$ et une complexité temporelle en $\boxed{\mathcal{O}(n^3).}$

5. Il suffit de nouveau de raisonner par récurrence avec l'hypothèse $\mathscr{P}(k)$ « Pour tous $i, j \in [\![0, n-1]\!]$, le booléen $b_{i,j}^{(k)}$ correspond à l'existence d'un chemin reliant le sommet i au sommet j en ne passant que par des sommets parmi $0, 1, \ldots, k-1$ ».

   Initialisation $\mathscr{P}(0)$ est vraie vu la définition de $B^{(0)}$ (on va directement de i à j).

   Hérédité Soit $k \in [\![0, n-1]\!]$ tel que $\mathscr{P}(k)$ est vraie, et $i, j \in [\![0, n-1]\!]$.

   Alors il existe un chemin de i vers j en ne passant que des sommets parmi 0, 1, …, k à deux conditions :

- soit parce qu'il existait déjà un chemin de i vers j en ne passant que des sommets parmi $0, 1, \ldots, k-1$, c'est-à-dire que $b_{i,j}^{(k)}$ est vrai par hypothèse de récurrence,
- soit parce qu'il existe un chemin de i vers k en ne passant que des sommets parmi $0, 1, \ldots, k-1$, et un chemin de k vers j en ne passant que des sommets parmi $0, 1, \ldots, k-1$, c'est-à-dire que $b_{i,k}^{(k)}$ et $b_{k,j}^{(k)}$ sont vrais par hypothèse de récurrence.

Ainsi, on a bien $b_{i,j}^{(k+1)} = b_{i,j}^{(k)}$ ou $\left(b_{i,k}^{(k)} \text{ et } b_{k,j}^{(k)}\right)$ est le booléen « il existe un chemin de i vers j en ne passant que des sommets parmi 0, 1, …, k » ce qui établit la récurrence.

Finalement, pour tous $i, j \in [\![0, n-1]\!]$, $b_{i,j}^{(n)}$ correspond bien à l'assertion « Il existe un chemin reliant le sommet i au sommet j ».

Python

```python
def Warshall(M):
    n = len(M)
    B = [[M[i][j] != inf for j in range(n)] for i in range(n)]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                B[i][j] = B[i][j] or (B[i][k] and B[k][j])
    return B
```

Deuxième partie

Le problème du sac à dos

1. En notant ℓ et p la longueur et le poids d'une corde, on remarque que
   - $2p(D) = p(C)$ et $2\ell(D) > \ell(C)$ donc on ne choisira aucune corde C,
   - $2p(B) = p(A)$ et $2\ell(B) < \ell(A)$ donc on choisira au plus une corde B,
   - il est toujours préférable de prendre une corde A avec les contraintes ci-dessus.

   Afin de ne pas dépasser 10 kg, il y a alors deux possibilités :
   - soit une corde A et deux cordes D, ce qui donne une longueur de 48 m,
   - soit une corde A et une corde B, soit une longueur de 44 m.

   On choisit donc $\boxed{\text{une corde } A \text{ et deux cordes } D.}$

2. Il est toujours préférable de prendre au moins deux cordes et on ne peut pas dépasser 3 cordes compte tenu des contraintes.
   Avec trois cordes, l'optimum est atteint pour des cordes de poids total maximal, soit une corde A et une corde C pour une longueur de 46 m, et la seule possibilité pour trois cordes est de prendre les cordes B, C et D pour une longueur de 39 m.
   On choisit donc $\boxed{\text{une corde } A \text{ et une corde } C.}$

A. Sac à dos sans répétitions

3. Si $w = 0$, la capacité du sac est nulle, donc $\boxed{s(0, j) = 0.}$

   Si $j = 0$, on ne prend pas d'objet, donc $\boxed{s(w, 0) = 0.}$

4. Soit $j \geqslant 0$. Dans $s(w', j)$ on est sûr de ne pas avoir pris l'objet j, donc il y a deux possibilités :
   - soit on prend l'objet j, ce qui n'est possible que si $w \geqslant p_j$, et on obtient, avec cette hypothèse, la valeur optimale $s(w - p_j, j) + v_j$,
   - soit on ne prend pas l'objet j et on obtient, avec cette hypothèse, la valeur optimale $s(w, j)$.

   On en déduit que

$$\boxed{s(w, j+1) = \begin{cases} \max(s(w - p_j, j) + v_j, s(w, j)) & \text{si } w \geqslant p_j, \\ s(w, j) & \text{sinon.} \end{cases}}$$

5. Une version avec une matrice :

```python
def sac_a_dos(W, p, v):
    n = len(p)
    s = [[0 for _ in range(n + 1)] for _ in range(W + 1)]
    for j in range(n):
        for w in range(1, p[j]):
            s[w][j + 1] = s[w][j]
        for w in range(p[j], W + 1):
            max_avec_j = s[w - p[j]][j] + v[j]
            s[w][j + 1] = max(s[w][j], max_avec_j)
    return s[W][n]
```

On peut aussi l'écrire avec une liste à une dimension `s_j` qui à la fin de l'étape `j` va contenir

$$[s(0, j),\ s(1, j),\ \ldots,\ s(W, j)].$$

```python
def sac_a_dos(W, p, v):
    n = len(p)
    s_j = [0 for _ in range(W + 1)]
    for j in range(n):
        s_j_moins_1 = s_j.copy()
        # On ne met à jour que lorsque w >= p[j]
        for w in range(p[j], W + 1):
            max_avec_j = s_j_moins_1[w - p[j]] + v[j]
            if max_avec_j > s_j_moins_1[w]:
                s_j[w] = max_avec_j
    return s_j[W]
```

6. Par exemple, avec la version matrice :

```python
def sac_a_dos(W, p, v):
    n = len(p)
    s = [[0 for _ in range(n + 1)] for _ in range(W + 1)]
    utilise = {}
    for j in range(n):
        for w in range(1, p[j]):
            s[w][j + 1] = s[w][j]
        for w in range(p[j], W + 1):
            max_avec_j = s[w - p[j]][j] + v[j]
            s[w][j + 1] = max(s[w][j], max_avec_j)
            if s[w][j + 1] == max_avec_j:
                utilise[(w, j + 1)] = None
    return s[W][n], utilise
```

7. Il s'agit du principe habituel de rétro propagation pour reconstituer la solution optimale.

```python
def retrouve_solution(W, p, utilise):
    objets_utilises = []
    n = len(p)
    w = W # poids restant
    for j in range(n - 1, -1, -1):
        if (w, j + 1) in utilise:
            objets_utilises.append(j)
            w -= p[j]
    return objets_utilises
```

8. La complexité temporelle de la fonction `sac_a_dos` est $\boxed{O(nW)}$ avec les notations du sujet, directement avec les deux boucles imbriquées.
La complexité spatiale est $\boxed{O(nW)}$ avec la version matrice, et $\boxed{O(W)}$ avec la version vecteur.

9. L'avantage d'une version récursive avec mémoïsation est qu'elle ne nécessite pas de remplir l'intégralité du tableau : on part des données initiales et on descend jusqu'à un cas de base seulement puis on remonte en remplissant le tableau pour les cas rencontrés seulement. Si toutes les données sont multiples de 100, on ne remplira que des cases sur des lignes multiples de 100.

Python

```python
def sac_a_dos2(W, p, v):

    dico = {}

    def s(w, j):
        if (w, j) not in dico:
            if 0 in (w, j):
                resultat = 0
            elif w < p[j - 1]:
                resultat = s(w, j - 1)
            else:
                resultat = max(s(w - p[j - 1], j - 1) + v[j - 1], s(w, j - 1))
            dico[(w, j)] = resultat
        return dico[(w, j)]

    return s(W, len(p))

```

B. Sac à dos avec répétitions

10. On a $\boxed{k(w) = \max\{k(w - w_j) + v_j \ ; \ 1 \leqslant j \leqslant n \text{ tel que } w_j \leqslant w\}.}$

11. Version tabulaire (bas en haut).

Python

```python
def sac_a_dos_repetition(W, p, v):
    k = [0 for _ in range (W + 1)]
    n = len(p)
    for w in range(1, W + 1):
        L = [k[w - p[j]] + v[j] for j in range(n) if p[j] <= w]
        k[w] = max(L) if L != [] else 0
    return k[W]
```

12. La complexité temporelle est $\boxed{O(nW)}$ (boucles sur `w` et sur `j`), la complexité spatiale est $\boxed{O(W).}$

C. Autres variantes

13. Il suffit que chaque objet apparaisse autant de fois que son nombre dans `p` et `v`.