Informatique : corrigé du devoir n°7

Exercice 1

1)

```
def rendu(montant):
    billets = [500, 200, 100, 50, 20, 10, 5, 2, 1]
    L = []
    for billet in billets:
        L.append(montant // billet)
        montant = montant % billet
    return L
```

2) Si la liste des pièces est (1,3,4) alors pour un montant de 6 euros l'algorithme glouton rendra une pièce de 4 et deux pièces de 1, alors qu'on peut rendre deux pièces de 3. Cet algorithme n'est donc pas optimal.

Exercice 2

1) Version longue :

```
def tableau_nul(n, p):
    t = []
    for i in range(n):
        ligne = []
        for j in range(p):
            ligne.append(0)
        t.append(ligne)
    return t
```

Version courte :

```
def tableau_nul(n, p):
    return [[0]*p for i in range(n)]
```

2) On peut itérer sur le chemin et récupérer directement les couples de coordonnées en faisant `for i, j in chemin` :

```
def poids(chemin, t):
    p = 0
    for i, j in chemin:
        p = p + t[i][j]
    return p
```

Version courte :

```
def poids(chemin, t):
    return sum(t[i][j] for i, j in chemin)
```

2) Le chemin de poids maximal de ce tableau est `[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3)]`. Son poids est 28.

3) a) L'algorithme glouton donne le chemin `[(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3)]` dont le poids est 21. Il n'est donc pas optimal.

b) `n` est le nombre de lignes du tableau, `p` le nombre de colonnes. `i` et `j` sont les coordonnées du point où on se trouve.

c)

`# 1` : Tant qu'on n'est pas arrivé à la case en bas à droite on continue.

`# 2` : Si on est sur la dernière ligne, on va vers la droite.

`# 3` : Si on est sur la dernière colonne, on va vers le bas.

`# 4` : Sinon on va vers la case dont le poids est le plus grand.

d) La complexité asymptotique est clairement $O(n+p)$ car on fait $n+p-2$ déplacements et qu'à chaque déplacement on fait au maximum une dizaine d'opérations élémentaires. Plus précisément :

- La comparaison dans le `while` est toujours effectuée, ainsi que les soustractions `n-1` et `p-1`.
- Il y a toujours une affectation et une addition `i = i + 1` ou `j = j + 1`.

– La comparaison `i == n - 1` est toujours effectuée, `j == p - 1` aussi sauf quand on est tout en bas et la comparaison des cases avec les accès au tableau est effectuée sauf si on est tout en bas ou tout à droite.

– Le `append` est toujours effectué.

Le nombre précis d'opérations dépend du fait qu'on arrive tout en bas ou tout à droite rapidement ou pas.

4) a) Le couple $(2, 3)$ correspond au chemin `[(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3)]`.

b)

```
def creer_chemin(n, p, deplacements_vers_le_bas):
    i, j = 0, 0
    chemin = [(0, 0)]
    for k in range(1, n+p-1):
        if k in deplacements_vers_le_bas:
            i = i + 1
        else:
            j = j + 1
        chemin.append((i, j))
    return chemin
```

c)

```
def optimal(t):
    n , p = len(t), len(t[0])
    poids_max = 0
    for combinaison in combinations(range(1, n+p-1), n-1):  # choix de n-1 entiers entre 1 et n+p-2
        chemin = creer_chemin(n, p, combinaison)
        pds = poids(chemin, t)
        if pds > poids_max:
            poids_max = pds
            chemin_optimal = chemin
    return chemin_optimal
```

d) Construire un chemin revient à choisir les étapes des $n-1$ déplacements vers la bas parmi les $n+p-2$ déplacements. Il y a donc $\binom{n+p-2}{n-1}$ chemins.

Pour $n = p = 20$ cela donne $\binom{38}{19} = 35345263800$ chemins possibles. La fonction `optimal` ne peut pas traiter autant de cas en un temps raisonnable !

5) a) Le tableau des poids maximaux associé à $\begin{pmatrix} 2 & 5 & 6 & 7 \\ 5 & 1 & 6 & 0 \\ 5 & 2 & 8 & 1 \end{pmatrix}$ est $\begin{pmatrix} 2 & 7 & 13 & 20 \\ 7 & 8 & 19 & 20 \\ 12 & 14 & 27 & 28 \end{pmatrix}$.

b) On remplit sans difficulté la première colonne et la première ligne (il suffit d'additionner les entiers rencontrés). Ensuite on parcourt le reste du tableau de gauche à droite et de haut en bas et on utilise le fait que le poids maximal d'un chemin arrivant à la case de coordonnées (i, j) s'obtient en additionnant le nombre situé dans cette case soit au poids maximal d'un chemin arrivant à la case située au-dessus de cette case (donc de coordonnées $(i-1, j)$), soit au poids maximal d'un chemin arrivant à la case située à sa gauche (donc de coordonnées $(i, j-1)$).

```
def tableau_poids_max(t):
    n, p = len(t), len(t[0])
    tab = [[0]*p for i in range(n)]        # tableau rempli de 0
    tab[0][0] = t[0][0]
    for i in range(1, n):                  # on remplit d'abord la première colonne
        tab[i][0] = tab[i-1][0] + t[i][0]
    for j in range(1, p):                  # et la première ligne
        tab[0][j] = tab[0][j-1] + t[0][j]
    for i in range(1, n):                  # puis le reste du tableau
        for j in range(1, p):
            tab[i][j] = max(tab[i-1][j]+t[i][j], tab[i][j-1]+t[i][j])
    return tab
```

Ou, en utilisant le fait que `tab[-1][j]` et `tab[i][-1]` sont nuls au départ :

```
def tableau_poids_max(t):
    n, p = len(t), len(t[0])
    tab = [[0]*p for i in range(n)]
```

```
    for i in range(n):
        for j in range(p):
            tab[i][j] = max(tab[i-1][j], tab[i][j-1]) + t[i][j]
    return tab
```

c) On construit le chemin à l'envers, en partant de la dernière case et en regardant à chaque case si, quand on a construit le tableau des poids maximaux, on est venu du haut ou de la gauche. Quand on arrive sur la première case, il n'y a plus qu'à renvoyer le résultat à l'envers.

```
def dynamique(t):
    n, p = len(t), len(t[0])
    tab = tableau_poids_max(t)
    i , j = n-1, p-1                    # on part de la dernière case
    chemin = [(i, j)]
    while (i, j) != (0, 0):
        if i == 0:                      # si on est sur la première ligne on va vers la gauche
            j = j - 1
        elif j == 0:                    # si on est sur la première colonne on va vers le haut
            i = i - 1
        elif tab[i-1][j] > tab[i][j-1]: # sinon on va vers la case voisine la plus grande
            i = i - 1
        else:
            j = j - 1
        chemin.append((i, j))
    return chemin[::-1]                 # on renvoie le chemin à l'envers
```