Informatique : devoir n°7 (non surveillé)

Un **algorithme glouton** est un algorithme où, à chaque étape, on fait le choix local optimal.

Considérons par exemple le problème du voyageur de commerce qui doit visiter une liste de villes en essayant de parcourir le moins de distance possible. L'algorithme glouton associé consiste à visiter à chaque étape la ville non visitée la plus proche.

Un algorithme glouton n'est pas toujours optimal, mais il permet généralement d'obtenir une solution de bonne qualité en un temps raisonnable.

Exercice 1 - Problème du rendu de monnaie

Les différentes valeurs des billets et des pièces en euros sont : 500, 200, 100, 50, 20, 10, 5, 2, 1. Le problème du rendu de monnaie est le suivant : on veut rendre un montant entier d'argent en utilisant le plus petit nombre possible de billets et de pièces.

Un algorithme glouton consiste à donner systématiquement le billet de plus grande valeur qui ne dépasse pas la somme restante. Par exemple, pour 29 euros, on commence par donner un billet de 20 euros, puis un billet de 5 et enfin deux pièces de 2.

1) Écrire une fonction `monnaie` qui, recevant un montant entier d'argent, renvoie une liste dont les éléments correspondent aux nombres de billets ou de pièces de chaque valeur rendus. Par exemple, `monnaie(29)` doit renvoyer la liste `[0, 0, 0, 0, 1, 0, 1, 2, 0]` (soit zéro billet de 500, 200, 100, 50, un billet de 20, zéro billet de 10, un billet de 5, deux pièces de 2 et zéro pièce de 1).

2) On peut démontrer (mais c'est assez difficile) que, lorsque la liste des pièces est $(1, 2, 5)$, cet algorithme glouton est optimal. Est-ce encore le cas lorsque la liste des pièces est $(1, 3, 4)$? On pourra considérer par exemple un montant à rendre de 6 euros.

Exercice 2 - Chemin de poids maximal dans un tableau bidimensionnel

Dans cet exercice on appelle tableau une liste de listes de même longueur. Par exemple, le tableau $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ est représenté par la liste `[[1, 2, 3], [4, 5, 6]]`.

On notera que si `t` est un tableau représenté de cette manière, alors `t[i]` est la i^e ligne de `t` et donc `t[i][j]` est le j^e terme de la i^e ligne. Ainsi, si `t` est le tableau ci-dessus, alors `t[0][0]` vaut `1`, `t[1][0]` vaut `4`, `t[0][1]` vaut `2`, etc.

On considère des chemins dans un tableau d'entiers positifs qui partent de la case en haut à gauche pour arriver à celle en bas à droite en n'allant que vers la droite ou vers le bas. On représentera un tel chemin par une liste de couples qui correspondent aux coordonnées des cases traversées. Par exemple, pour le tableau ci-dessus, le chemin `[(0, 0), (0, 1), (1, 1), (1, 2)]` est représenté en gras ci-dessous :

$$\begin{pmatrix} \mathbf{1} & \mathbf{2} & 3 \\ 4 & \mathbf{5} & \mathbf{6} \end{pmatrix}$$

On appelle poids du chemin la somme des entiers rencontrés au cours du déplacement (dans l'exemple précédent, le poids du chemin est $1 + 2 + 5 + 6 = 14$).

1) Écrire une fonction `poids` qui, recevant un tableau et un chemin, renvoie le poids de ce dernier.

```
>>> poids([[1, 2, 3], [4, 5, 6]], [(0, 0), (0, 1), (1, 1), (1, 2)])
14
```

Un tableau étant donné, on souhaite déterminer un chemin dont le poids est maximal. Par exemple, pour le tableau précédent, il s'agit du chemin `[(0, 0), (1, 0), (1, 1), (1, 2)]` dont le poids est 16.

2) Quel est le chemin de poids maximal du tableau $\begin{pmatrix} 2 & 5 & 6 & 7 \\ 5 & 1 & 6 & 0 \\ 5 & 2 & 8 & 1 \end{pmatrix}$?

3) Dans un premier temps on propose d'utiliser un algorithme glouton : à chaque étape on se dirige vers la case voisine (en bas ou à droite) la plus grande (si les deux sont égales on choisit celle de droite).

a) Quel chemin obtient-on avec cet algorithme pour le tableau de la question précédente ? Ce chemin est-il optimal ?

On propose de programmer cet algorithme de la manière suivante :

```
def glouton(t):
    n, p = len(t), len(t[0])
    i, j = 0, 0
    chemin = [(0, 0)]
    while (i, j) != (n-1, p-1):      # 1
        if i == n - 1:               # 2
            j = j + 1                # 2
        elif j == p - 1:             # 3
            i = i + 1                # 3
        elif t[i+1][j] > t[i][j+1]:  # 4
            i = i + 1                # 4
        else:                        # 4
            j = j + 1                # 4
        chemin.append((i, j))
    return chemin
```

b) Que représentent les variables `n`, `p`, `i` et `j` ?

c) Expliquer rapidement les parties `# 1`, `# 2`, `# 3` et `# 4` de la fonction.

d) Déterminer le nombre d'opérations élémentaires (additions, affectations, comparaisons et accès au tableau) effectuées par la fonction quand on l'applique à un tableau à n lignes et p colonnes.

4) L'algorithme glouton ne renvoyant pas toujours le chemin optimal, on se propose de déterminer celui-ci par force brute, en testant tous les chemins possibles.

Un chemin dans un tableau à n lignes et p colonnes est composé de $n+p-2$ déplacements : $n-1$ vers le bas et $p-1$ vers la droite. Pour définir un chemin il suffit de donner les numéros des déplacements vers le bas. Par exemple, avec le tableau de la question 2, le couple $(1,4)$ signifie qu'on se déplace d'abord vers le bas (étape 1), puis deux fois vers la droite (étapes 2 et 3), puis vers le bas (étape 4) et enfin vers la droite (étape 5), ce qui correspond au chemin `[(0, 0), (1, 0), (1, 1), (1, 2), (2, 2), (2, 3)]` :

$$\begin{pmatrix} \mathbf{2} & 5 & 6 & 7 \\ \mathbf{5} & \mathbf{1} & \mathbf{6} & 0 \\ 5 & 2 & \mathbf{8} & \mathbf{1} \end{pmatrix}$$

a) Avec ce même tableau, quel chemin est défini par le couple $(2,3)$?

b) Écrire une fonction `creer_chemin` qui, recevant deux entiers n et p correspondant aux dimensions du tableau et un tuple de longueur $n-1$ correspondant aux déplacements vers le bas, renvoie le chemin associé.

```
>>> creer_chemin(3, 4, (1, 4))
[(0, 0), (1, 0), (1, 1), (1, 2), (2, 2), (2, 3)]
```

Pour générer tous les chemins possibles il suffit donc de générer tous les $(n-1)$-uplets de l'ensemble $\{1,\dots,n+p-2\}$. Pour cela on peut utiliser la fonction `combinations` du module `itertools` :

```
>>> from itertools import combinations
>>> list(combinations(range(1, 6), 2))
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4),
 (3, 5), (4, 5)]
```

Il ne reste plus qu'à rechercher parmi les chemins ainsi générés celui dont le poids est maximal.

c) Écrire une fonction `optimal` qui, recevant recevant un tableau `t`, renvoie un chemin de poids maximal associé à `t`.

```
>>> optimal([[1, 2, 3], [4, 5, 6]])
[(0, 0), (1, 0), (1, 1), (1, 2)]
```

d) Combien y a-t-il de chemins dans un tableau à n lignes et p colonnes ? Faire le calcul pour $n=p=20$ (on peut utiliser la calculatrice ou la fonction `factorial` du module `math`). La fonction `optimal` est-elle utilisable dans ce cas ?

5) On propose enfin la méthode suivante (programmation dynamique) : on commence par construire un tableau de même taille que `t` dont chaque case contient le poids d'un chemin maximal arrivant à cette case. Ainsi le tableau des poids maximaux associé au tableau $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ est $\begin{pmatrix} 1 & 3 & 6 \\ 5 & 10 & 16 \end{pmatrix}$.

a) Quel est le tableau des poids maximaux associé à $\begin{pmatrix} 2 & 5 & 6 & 7 \\ 5 & 1 & 6 & 0 \\ 5 & 2 & 8 & 1 \end{pmatrix}$?

b) Écrire une fonction `tableau_poids_max` qui, recevant un tableau `t`, renvoie le tableau des poids maximaux associé à `t`.

```
>>> tableau_poids_max([[1, 2, 3], [4, 5, 6]])
[[1, 3, 6], [5, 10, 16]]
```

c) Une fois le tableau des poids maximaux créé, il est facile de déterminer le chemin de poids maximal en partant de la dernière case et en "remontant". Écrire une fonction `dynamique` qui, recevant un tableau `t`, renvoie un chemin de poids maximal associé à `t`.

```
>>> dynamique([[1, 2, 3], [4, 5, 6]])
[(0, 0), (1, 0), (1, 1), (1, 2)]
```

Remarque : cette méthode permet d'obtenir un chemin optimal en temps raisonnable. Ainsi, pour un tableau à 1000 lignes et 1000 colonnes, la fonction `dynamique` renvoie le résultat en moins d'une seconde alors que la fonction `optimal` doit tester environ 5×10^{599} chemins...