TP n°4 : algorithmes gloutons

Objectifs

Découvrir la notion d'algorithme glouton, et manipuler des matrices.

1 Manipulations sur les matrices pour s'échauffer

La matrice $A = \begin{pmatrix} 1 & 7 & 9 & 2 \\ 8 & 6 & 3 & 2 \\ 1 & 6 & 7 & 8 \\ 2 & 9 & 8 & 2 \end{pmatrix}$ sera représentée en Python par une liste de listes : `A = [[1,7,9,2], [8,6,3,2], [1,6,7,8],[2,9,8,2]]`.

Ainsi `A[0][1]` est l'élément de la ligne d'indice 0 et de la colonne d'indice 1, soit le nombre 7. De même, `A[1][2]` vaut 3.

Répondre aux questions suivantes à l'aide de Python.

1. Calculer la somme des coefficients de A.
2. Calculer la somme des coefficients diagonaux de A (ceux dont l'indice de ligne est le même que l'indice de colonne).
3. Calculer la somme des coefficients strictement au-dessus de la diagonale.

2 Minimum Cost Path Problem

Objective : Given a 2D-matrix where each cell has a cost to travel. You have to write an algorithm to find a path from left-top corner to bottom-right corner with minimum travel cost. You can move only right or down.

You can see an example with the matrix A in the following picture.

2.1 Stratégie gloutonne

Nous décrivons ici une stratégie appelée «méthode gloutonne» : elle consiste à chaque étape à choisir comme nouvelle case (case adjacente à droite ou en bas) celle dont le poids est minimal.

1. Décrire le chemin choisi par la méthode gloutonne pour la matrice $B = \begin{pmatrix} 2 & 3 & 4 & 5 & 1 & 2 \\ 10 & 2 & 7 & 9 & 2 & 3 \\ 11 & 5 & 2 & 3 & 6 & 6 \\ 20 & 8 & 10 & 5 & 7 & 8 \end{pmatrix}$.

FIGURE 1 – Chemin de poids minimal

2. Écrire une fonction `indice_suivant(A:[[int]], i:int, j:int)-> (int,int)` qui prend en argument une matrice A, des indices i et j et renvoie les indices de la case adjacente que l'on choisit, donc celle de poids minimal. Par exemple, `indice_suivant(A, 0, 1)` renvoie $(1, 1)$ et `indice_suivant(A, 1, 3)` renvoie $(2, 3)$.

   On pourra compléter le code suivant :

```
def indice_suivant(A:[[int]], i:int, j:int)-> (int,int):
    nbligne =
    nbcolonne =
    assert (i, j) != (nbligne -1, nbcolonne -1) # la case d'arrivée n'a pas d'indic

    if j == :     # colonne de droite
        return (..., ...)

    if  :          # ligne du bas
        return (..., ....)

    # cas général
    if A[...][...] > A[...][...]:
        return (..., ... )
    else:
        return (..., ...)
```

3. En déduire une fonction `glouton(A)` qui calcule le poids du chemin minimal dans une matrice A selon cette stratégie dite gloutonne.

4. La stratégie gloutonne fournit-elle toujours le chemin de poids minimal (on pourra raisonner avec une matrice carrée de taille 3 ?

2.2 Un exemple de programmation dynamique

On se propose ici de découvrir une méthode toujours opérante.

Pour cela on va construire une matrice auxiliaire C (qu'on appellera matrice de coût) de même format que A telle que son coefficient C_{ij} contienne la somme minimale des chemins allant de la case d'indice $(0,0)$ à la case d'indice (i,j).

5. Compléter la matrice de côut C associée à la matrice A.

$$C = \begin{pmatrix} 1 & 8 & & \\ 9 & 14 & & \\ & & & 27 \\ & 21 & & \end{pmatrix}$$

6. Déterminer une relation de récurrence vérifiée par les coefficients C_{ij} (on distinguera plusieurs cas, notamment si la case est sur un bord de la matrice).

7. Compléter la fonction `cout_mini(A)` qui retourne la matrice de coût C associée à une matrice A retourne et la longueur du chemin minimal.

```
def cout_mini(A:[[int]])-> int:
    m, n = len(A), len(A[0]) # nbre de lignes et  de colonnes
    C = [[0 for i in range(m)] for j in range(n)] # matrice nulle de taille (m, n)
    C[0][0] = A[0][0]
    # on remplit la première ligne
    for j in range(...,...):
        C[0][j] = C[0][j-1] + A[...][...]
    # on remplit la première colonne
    for i in range(...,...):
        C[i][0] = .... + ....
    # on remplit le reste du tableau
    for i in range(..., ...):
        for j in range(..., ...):
            C[i][j] = min(...., ....) + .....
    return ..............
```

3 Rendu de monnaie

On suppose que l'on vit dans un monde imaginaire où l'on dispose de pièces de monnaies en quantité illimitée dont la valeur en euros est décrite par la liste `p = [2,3,11,13]`.

Pour payer une somme de 15 euros, on cherche à utiliser un nombre minimal de pièces du système p. On peut par exemple utiliser :

- soit 6 pièces de deux euros et 1 pièce de trois euros, ce qui donne 7 pièces au total
- soit 2 pièces de deux euros, 1 pièce de onze euros, ce qui donne 3 pièces au total

- soit 1 pièces de deux euros et 1 pièce de treize euros, ce qui donne 2 pièces au total

Il y a d'autres possibilités que nous n'avons pas écrites, mais le nombre minimal de pièces pour payer 15 euros est de 2 (1 pièce de deux euros et 1 pièce de treize euros).

Si $s \geqslant 2$ est un entier, on note $f(s)$ le nombre minimal de pièces du système `p` que l'on peut utiliser pour payer une somme de s euros. Par exemple $f(15) = 2$.

Le but de cette section est d'écrire un algorithme calculant $f(s)$ [1].

Une stratégie que l'on appelle «gloutonne» en informatique consiste à essayer d'utiliser en premier la pièce avec le montant le plus important, puis de réitérer avec le montant restant. Par exemple avec $s = 60$, la technique gloutonne consisterait à utiliser dans l'ordre :

- on utilise 4 pièces de 13 euros, cela fait 52 euros, il reste 8 euros à payer.
- pour payer ces 8 euros, on ne peut pas utiliser de pièces de 11 euros. il reste encore 8 euros à payer.
- pour payer ces 8 euros, on utilise 2 pièces de 3 euros, ce qui fait 6 euros, il reste donc 2 euros à payer.
- pour payer ces 2 euros, on utilise 1 pièce de 2 euros. On a terminé

Nous avons avec cette stratégie gloutonne utilisé 7 pièces pour payer 60 euros.

1. La stratégie «gloutonne» donne-t-elle toujours le nombre minimal de pièces ? Proposer ensuite une valeur de s pour lequel la stratégie gloutonne ne trouve pas de solution.
2. Compléter la fonction suivante `glouton(s:int, p:[int])-> int` qui renvoie le résultat de la stratégie gloutonne lorsque que l'on veut payer s euros, avec un minimum de pièces du système p.

```
def glouton(s:int, p:[int])-> int:
    n = len(p)
i = n -1 # indice de la pièce la plus grande
    nbre_pieces_utilisees = 0
    while s ..... and i.....: # tant qu'il reste....
        if ......:
            nbre_pieces_utilisees += 1  # on utilise la pièce i
            s = ...... # nouvelle valeur de s
        else:
            i = i - 1 # on utilise la pièce de valeur inférieure
    # arrivé ici, soit s = 0, soit i < 0
    if s .....:
        return nbre_pieces_utilisees
    else:
        return None # l'algo glouton ne trouve pas de solution.
```

3. On vérifiera que la complexité de la fonction `glouton(S, p)` est en $O(S+n)$ où n est la longueur de p (on pourra constater que la quantité $s+i$ décroît à chaque itération...)

1. On peut démontrer sans trop de difficultés, en regardant les différentes valeurs de s modulo 3, que pour tout entier $s \geqslant 2$, il est possible de payer la somme s uniquement à l'aide des pièces de deux et trois euros, et donc en particulier avec des pièces de la liste p.