Ce devoir est constitué de trois exercices : les deux premiers sont à faire en Python, le dernier en SQL.

I Polynôme et dictionnaire

Soit $P = \sum_{k=0}^{n} a_k X^k$ un polynôme. On représente P par un dictionnaire `p` tel que, pour tout $k \in [\![0, n]\!]$, si $a_k \neq 0$ alors `p[k]` vaut a_k (on ne stocke pas les coefficients nuls de P). Dit autrement, `p[k]` contient le coefficient de degré k de P.

1. Définir le dictionnaire représentant le polynôme $7 + 3X^2 - X^5$.

> Solution : `p = {0 : 7, 2 : 3, 5 : -1}`

2. Écrire une fonction `degre` renvoyant le degré d'un polynôme.

> Solution :

```python
def degre(p):
    maxi = 0
    for k in p:
        if k > maxi:
            maxi = k
    return maxi
```

> Ou, une solution un peu plus rapide à écrire :

```python
def degre(p):
    return max(p.keys())
```

3. Écrire une fonction `derive` qui renvoie la dérivée P' d'un polynôme P.

> Solution :

```python
def derive(p):
    dp = {}
    for k in p:
        if k != 0: # le terme constant disparaît
            dp[k - 1] = p[k]
    return dp
```

4. Définir une fonction `somme(p, q)` renvoyant un dictionnaire représentant la somme des deux polynômes `p` et `q`. Quelle est sa complexité en fonction des degrés n_1 et n_2 de `p` et `q` ?

> Solution : La complexité de la fonction suivante est $\mathrm{O}(n_1 + n_2)$ (en considérant que les opération de dictionnaire sont en $\mathrm{O}(1)$, ce qui est normalement le cas en moyenne seulement).

```python
def somme(p, q):
    r = {}
    for k in p:
        r[k] = p[k]
    for k in q:
        r[k] += q[k]
    for k in r:
        if r[k] == 0:
            del r[k] # pour éviter les termes nuls
    return r
```

5. Faire de même avec une fonction `produit(p, q)`.

> Solution : Chaque terme $a_i X^i$ de P multiplié avec un terme $b_j X^j$ de Q va donner un terme $a_i b_j X^{i+j}$ dans le produit.

D'où :

```python
def produit(p, q):
    r = {}
    for i in p:
        for j in q:
            if i + j in r:
                r[i + j] += p[i]*q[j]
            else:
                r[i + j] = p[i]*q[j]
    return r
```

La complexité est clairement $\mathrm{O}(n_1 \times n_2)$. On pourrait aussi enlever les termes nuls comme à la question précédente.

6. Écrire une fonction `evalue(x, p)` renvoyant $P(x)$, si possible avec $\mathrm{O}(n)$ multiplications, où $n = \deg(P)$.

Solution : On rappelle que le calcul de `x**n` demande $\mathrm{O}(\log(n))$ multiplications (avec l'algorithme d'exponentiation rapide). Pour avoir une complexité $\mathrm{O}(n)$, on peut utiliser la méthode de Horner. Une autre possibilité est de calculer et stocker toutes les puissances, pour éviter de les recalculer en totalité :

```python
def evalue(x, p):
    n = degre(p)
    puissances = [1]
    for i in range(n):
        puissances.append(puissances[-1]*x)
    res = 0
    for k in p:
        res += p[k]*puissances[k]
    return res
```

II Chemin dans une matrice

Étant donnée une matrice d'entiers $M = (a_{i,j})$ de taille $n \times k$, on veut connaître un chemin (n'utilisant que des déplacements $\rightarrow$ ou $\downarrow$) de la case en haut à gauche (de coordonnées $(0,0)$) à la case en bas à droite (de coordonnées $(n-1, k-1)$) maximisant la somme des entiers rencontrés (le **poids** du chemin).

1. Quelle serait la complexité d'un algorithme de recherche exhaustive, énumérant tous les chemins possibles de $(0,0)$ à $(n-1, n-1)$? (on suppose pour simplifier que $n = k$, dans cette question)

Solution : Un chemin de $(0,0)$ à $(n-1, n-1)$ doit effectuer $n-1$ déplacements vers le bas ($\rightarrow$) et $n-1$ vers la droite ($\downarrow$).
Parmi ces $2n-2$ déplacements, il suffit, pour déterminer le chemin, de choisir ceux qui sont $\rightarrow$ (les $\downarrow$ sont alors déterminés). Il y a donc $\boxed{\binom{2n-2}{n-1}}$ choix possibles.
En posant $p = n-1$ et en utilisant la formule de Stirling :

$$\binom{2n-2}{n-1} = \binom{2p}{p} = \frac{(2p)!}{(p!)^2} \sim \ldots \sim \frac{2^{2p}}{\sqrt{p\pi}}$$

La complexité d'un tel algorithme serait donc exponentielle...

2. Supposons qu'un chemin C de poids maximum de $(0,0)$ à $(n-1, k-1)$ passe par la case (i,j). Montrer que le sous-chemin de C de $(0,0)$ à (i,j) est de poids maximum (c'est une propriété de **sous-optimalité**).

Solution : Supposons par l'absurde qu'il existe un chemin de $(0,0)$ à (i,j) de poids strictement supérieur. Alors en concaténant ce chemin avec la partie de C de (i,j) à $(n-1, k-1)$, on contredirait la maximalité de C : c'est absurde.

3. Soit $p_{i,j}$ le poids maximum d'un chemin de $(0,0)$ à (i,j). Donner, en la prouvant, une formule de récurrence sur $p_{i,j}$ pour $i > 0$ et $j > 0$.

> Solution : Un chemin de poids maximum jusqu'à $(i, j) \neq (0, 0)$ passe nécessairement par $(i-1, j)$ ou $(i, j-1)$: d'après la question 2, son poids est donc soit $p_{i-1,j} + a_{i,j}$, soit $p_{i,j-1} + a_{i,j}$. D'où :
>
> $$\boxed{p_{i,j} = \max(p_{i-1,j}, p_{i,j-1}) + a_{i,j}}$$

4. En déduire une fonction récursive simple `poids_max` tel que `poids_max(m, i, j)` renvoie le poids maximum d'un chemin de $(0, 0)$ vers (i, j) dans la matrice `m`. Que dire de sa complexité ?

> Solution : La complexité de cette fonction est exponentielle, comme expliqué en question 1.

```python
def poids_max(m, i, j):
    if i == 0 and j == 0: return m[0][0]
    if i == 0: return poids_max(m, 0, j-1) + m[0][j]
    if j == 0: return poids_max(m, i-1, 0) + m[i][0]
    return max(poids_max(m, i-1, j), poids_max(m, i, j-1)) + m[i][j]
```

5. Écrire une fonction `poids_max_dp(m)` donnant le poids maximum d'un chemin de la case en haut à gauche à la case en bas à droite dans la matrice `m`, en utilisant une méthode par programmation dynamique. Comparer sa complexité avec la méthode précédente.

> Solution : La complexité de la fonction suivante est bien meilleure : $\boxed{\mathrm{O}(nk)}$.

```python
def poids_max_dp(m):
    n, k = len(m), len(m[0])
    p = [[0 for _ in range(k)] for _ in range(n)] # matrice n*k remplie de 0
    p[0][0] = m[0][0]
    for i in range(1, n):
        p[i][0] = p[i - 1][0] + m[i][0]
    for j in range(1, k):
        p[0][j] = p[0][j - 1] + m[0][j]
    for i in range(1, n):
        for j in range(1, k):
            p[i][j] = max(p[i - 1][j], p[i][j - 1]) + m[i][j]
    return p[n - 1][k - 1]
```

> Remarque : Attention à traiter les cas où $i = 0$ et $j \neq 0$ (et cas symétrique), pour ne pas dépasser de la matrice.
> Autre solution, par mémoïsation (où on utilise un autre cas de base pour éviter de dépasser de la matrice) :

```python
def poids_max_memo(m):
    n, k = len(m), len(m[0])
    p = {}
    def aux(i, j):
        if (i, j) == (0, 0): return m[0][0]
        if i == -1 or j == -1: return 0
        if not (i, j) in p:
            p[(i, j)] = max(aux(i - 1, j), aux(i, j - 1)) + m[i][j]
        return p[(i, j)]
    return aux(n - 1, k - 1)
```

6. La fonction précédente ne donne que le poids maximum d'un chemin... Expliquer comment faire pour trouver un chemin de poids maximum.

> Solution : On peut utiliser une matrice `pere` de même taille que `m` pour stocker telle que `pere[i][j]` soit le couple (i', j') de la case précédent la case (i, j) dans un chemin de poids maximum de $(0, 0)$ à (i, j). On peut alors reconstruire le chemin en remontant la matrice `pere` depuis la case en bas à droite.

7. (à faire seulement si vous avez fini tout le reste) Écrire une fonction `chemin_max_dp(m)` renvoyant la liste des cases d'un chemin de poids maximum de $(0, 0)$ à $(n-1, k-1)$ dans la matrice `m`.

Solution :

```python
def chemin_max_dp(m):
    n, k = len(m), len(m[0])
    p = [[0 for _ in range(k)] for _ in range(n)]
    pere = [[(0, 0) for _ in range(k)] for _ in range(n)]
    p[0][0] = m[0][0]
    for i in range(1, n):
        p[i][0] = p[i-1][0] + m[i][0]
        pere[i][0] = (i - 1, 0)
    for j in range(1, k):
        p[0][j] = p[0][j - 1] + m[0][j]
        pere[0][j] = (0, j - 1)
    for i in range(1, n):
        for j in range(1, k):
            if p[i-1][j] > p[i][j - 1]:
                p[i][j] = p[i - 1][j] + m[i][j]
                pere[i][j] = (i-1, j)
            else:
                p[i][j] = p[i][j - 1] + m[i][j]
                pere[i][j] = (i, j - 1)

    # on reconstruit ensuite le chemin en partant de la fin
    chemin = [(n - 1, k - 1)]
    i, j = n-1, k-1
    while (i, j) != (0, 0):
        i, j = pere[i][j]
        chemin.append((i, j))
    return chemin[::-1] # inverse la liste
```

8. (à faire seulement si vous avez fini tout le reste) Soit G un graphe orienté pondéré acyclique (représenté par liste d'adjacence), s un sommet et $l = v_1, v_2, ..., v_n$ la liste de ses sommets dans un **ordre topologique**, c'est-à-dire que s'il y a une arête de v_i vers v_j dans G, alors $i < j$. En utilisant l, décrire un algorithme par programmation dynamique qui calcule les plus longs chemins dans G depuis s en temps $O(n + m)$, où m est le nombre d'arêtes de G.
Remarque : dans un graphe quelconque, et sous l'hypothèse P $\neq$ NP, il n'existe pas d'algorithme efficace pour trouver des plus longs chemins.

Solution : On peut utiliser une matrice `p` de taille n où `p[i]` est la longueur du plus long chemin dans G depuis s passant par le sommet v_i. On a alors la récurrence suivante :

$$p[i] = \begin{cases} 0 & \text{si } i = s \\ \max\limits_{j \in \text{prédécesseurs de } v_i} p[j] + w_{v_j, v_i} & \text{sinon} \end{cases}$$

où w_{v_j, v_i} est le poids de l'arête de v_j vers v_i dans G. On peut alors calculer les valeurs de `p` dans l'ordre de l en temps $\boxed{O(n + m)}$.
Remarque : on peut aussi prendre l'opposé $(\times - 1)$ des poids des arêtes et calculer les plus courts chemins avec Bellman-Ford (et pas Dijkstra qui ne marche pas avec des poids négatifs) qui donnerait une complexité O(nm).

III Base de donnée de corps célestes

A `SELECT masse FROM CORPS`

B.1

```sql
SELECT COUNT(DISTINCT id_corps)
FROM etat
WHERE datem < tmin()
```

B.2

```sql
SELECT id_corps, MAX(datem)
FROM etat
WHERE datem < tmin()
GROUP BY id_corps
```

B.3

```sql
SELECT masse,x,y,z,vx,vy,vz FROM corps AS c
JOIN etat AS e ON c.id_corps = e.id_corps
JOIN date_mesure AS d ON (datem = date_der AND e.id_corps = d.id_corps)
WHERE masse >= masse_min() AND ABS(x) < arete()/2
AND ABS(y) < arete()/2 AND ABS(z) < arete()/2
ORDER BY x*x+y*y+z*z
```