# Exercice 1 – requêtes dans une base de données

On cherche à interroger le système d'information des jeux olympiques. Le suivi des résultats aux jeux olympiques d'été de 1896 à 2021 est géré par la base de données JO.
Une partie des tables de cette base de données est présentée ci-dessous.

| Resultats | | | | |
|---|---|---|---|---|
| Id | EditionId | SportId | AthleteId | MedailleId |
| 1 | 1 | 17 | 7069 | 1 |
| 2 | 1 | 17 | 7655 | 2 |
| | | | | |
| 12 | 1 | 96 | 10580 | 3 |
| 13 | 1 | 96 | 18699 | 3 |
| 14 | 1 | 96 | 2516 | 1 |
| 15 | 1 | 96 | 7820 | 2 |
| | | | | |
| | | | | |

Cette table récapitule les différentes médailles distribuées lors des éditions successives des jeux olympiques d'été (avec l'identifiant de la médaille (clé primaire), le numéro de l'édition des jeux, l'identifiant du sport concerné, celui du sportif médaillé et l'identifiant du type de médaille).

| Athletes | | |
|---|---|---|
| Id | Nom | CodePays |
| 1 | AABYE, Edgar | ZZX |
| 2 | AALTONEN, Arvo Ossian | FIN |
| 3 | AALTONEN, Paavo Johannes | FIN |
| | | |
| 14 | ABALO, Luc | FRA |
| | | |

Cette table est celle de tous les athlètes participants, elle donne l'identifiant de l'athlète (clé primaire), son nom (prénom inclus), le code de son pays.

| Editions | | |
|---|---|---|
| Id | Ville | Annee |
| 1 | Athenes | 1896 |
| 2 | Paris | 1900 |
| 3 | Saint Louis | 1904 |
| | | |
| 32 | Tokyo | 2021 |

| Medailles | |
|---|---|
| Id | Metal |
| 1 | Or |
| 2 | Argent |
| 3 | Bronze |

Ces deux dernières tables identifient l'édition des jeux et celle du type de médaille.

Les différents attributs Id, EditionId, SportId, AthleteId, MedailleId et Annee sont de type integer ; Nom, Ville sont de type varchar(100) ; CodePays, Metal sont de type varchar(6).

1 ▷ Écrire en langage SQL une requête qui affiche les années où les jeux olympiques ont eu lieu à Athènes, depuis 1896.

2 ▷ Écrire en langage SQL une requête calculant le nombre de médailles remises à un athlète de 1896 à 2021.

3 ▷ Écrire en langage SQL une requête renvoyant la liste des médaillés français (CodePays FRA) aux premiers jeux olympiques répertoriés dans la base de donnée, classés par ordre alphabétique des noms.

4 ▷ Écrire en langage SQL une requête retournant l'année et le nombre de médailles remises à un athlète, pour chaque édition des jeux olympiques. Les résultats seront présentés par orde décroissant sur l'année.

5 ▷ Écrire en langage SQL une requête calculant le nombre de médailles remises à un athlète français (CodePays FRA) en 2008.

6 ▷ Écrire en langage SQL une requête renvoyant la liste des athlètes qui ont été médaillés à la fois en 2016 et en 2021 .

7 ▷ On suppose que l'on a construit la table suivante pour les jeux de 2008, à partir des données de la base JO, et qu'on l'a nommée TableauMedailles :

| TableauMedailles | | | | |
|---|---|---|---|---|
| Pays | Sport | Or | Argent | Bronze |
| | | | | |
| Espagne | cyclisme | 2 | 1 | 0 |
| | | | | |
| France | BMX | 1 | 1 | 0 |
| France | Escrime | 2 | 2 | 0 |
| | | | | |
| | | | | |

Écrire en langage SQL une requête renvoyant la liste des pays ayant obtenu au moins 10 médailles aux jeux de 2008, ainsi que leur nombre de médailles de chaque catégorie.

---

1 ▷
```sql
SELECT Annee FROM Editions WHERE Ville='Athenes'
```

2 ▷
```sql
SELECT COUNT(*) FROM Resultats
```

3 ▷
```sql
SELECT DISTINCT a.Nom
    FROM Resultats r JOIN Athletes a ON r.AthleteId=a.Id
    WHERE a.CodePays='FRA' AND r.EditionId=1
    ORDER BY a.Nom ASC
```

4 ▷
```sql
SELECT e.Annee, COUNT(*)
    FROM Resultats r JOIN Editions e ON r.EditionId=e.Id
    GROUP BY e.Annee
    ORDER BY e.Annee DESC
```

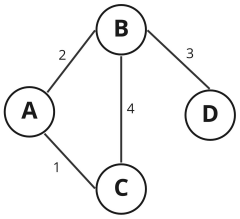
5 ▷
```sql
SELECT COUNT(*)
    FROM Resultats r JOIN Editions e ON r.EditionId=e.Id
                     JOIN Athletes a ON r.AthleteId=a.Id
```

6 ▷
```sql
(SELECT DISTINCT a.Nom
    FROM Athletes a JOIN Resultats r ON a.Id=r.AthleteId
                    JOIN Editions e ON r.EditionId=e.Id
    WHERE e.Annee=2016)
INTERSECT
(SELECT DISTINCT a.Nom
    FROM Athletes a JOIN Resultats r ON a.Id=r.AthleteId
                    JOIN Editions e ON r.EditionId=e.Id
    WHERE e.Annee=2021)
```

7 ▷
```sql
SELECT Pays, SUM(Or), SUM(Argent), SUM(Bronze)
    FROM TableauMedailles
    GROUP BY Pays
    HAVING SUM(Or) + SUM(Argent) + SUM(Bronze) >=10
```

## Exercice 2 – graphes et contournement d'obstacles

Considérons un graphe non orienté pondéré (on dit aussi valué) G. On rappelle que, dans ce contexte, la longueur d'un chemin est la somme des pondérations des arêtes du chemin. Par exemple, pour le graphe ci-dessous, le chemin «A → B → C» est de longueur 6 :




Notons qu'un tel graphe est décrit par sa liste d'adjacence et cette dernière peut-être représentée par le dictionnaire :

```
g = {'A': [('B',2), ('C',1)], 'B': [('A',2), ('C',4), ('D',3)], 'C': [('A',1), ('B',4)],
     'D': [('B',3)]}
```

Si l'on considère un sommet racine $r$ alors un algorithme bien connu pour déterminer, pour tout sommet $u$, le plus court chemin d'extrémités $r$ et $u$, est l'algorithme de Dijkstra dont on rappelle ci-dessous le pseudo-code :

**Dijkstra**(`G`, `r`)

**Entrée** : le graphe `G`, un sommet `r` de `G`

**Sortie** : dictionnaire `d` dont les clés sont les sommets et les valeurs sont les longueurs minimales correspondantes à partir de `r`,
dictionnaire `parent` donnant pour chaque sommet son parent dans le chemin le plus court venant de `r`

- marquer tous les sommets en *inconnu* sauf `r`
- marquer `r` en *vu*
- pour tout sommet `u` $\neq$ `r` : `d(u)` $\leftarrow +\infty$
- `d(r)` $\leftarrow 0$

∘ **TANT QUE** il existe au moins un sommet *vu* :

- soit `u` un sommet *vu* qui minimise `d`
- marquer `u` en *traité*

∘ **POUR** tout *successeur* `s` de `u` qui n'est pas *traité* :

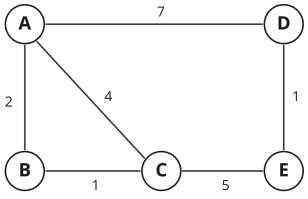
- calculer $\delta$ : somme de `d(u)` et du poids de l'arête `(us)`

∘ **SI** $\delta <$ `d(s)`

- `d(s)` $\leftarrow \delta$
- `parent(s)` $\leftarrow$ `u`
- marquer `s` en *vu*

∘ **Renvoyer** `d` et `parent`

**Q1 –** Appliquer l'algorithme de Dijkstra au graphe suivant avec pour origine le sommet A. On fera figurer les différentes étapes et l'on pourra par exemple rédiger la réponse à l'aide d'un tableau donnant les distances à A et les parents de chaque sommet.



**Q2 –** Considérons un graphe G et $r$ un sommet de G.
Compléter la fonction Python `vu_min(etat,d)` dont les entrées sont :

- un dictionnaire `etat` de clés les sommets de G et de valeurs une chaîne (`"inconnu"`, `"vu"` ou `"traite"`),
- et un dictionnaire `d` de mêmes clés et de valeurs des distances à partir du sommet $r$,

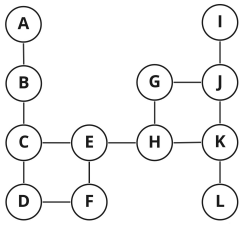
et qui renvoie le sommet `"vu"` le plus proche de $r$ (privilégier l'ordre lexicographique en cas d'égalité).
On rappelle que `float('+inf')` désigne le «nombre» $+\infty$.

```
def vu_min(etat, d):
    L = []
    d_min = float('+inf')

    << zone à compléter >>

    L.sort()  # classement dans l'ordre lexicographique
    return L[0]
```

**Q3 –** Compléter la fonction suivante qui programme l'algorithme de Dijkstra (il y a deux zones à compléter) :

```
def Dijkstra(gr, r):
    """
    Entrées : dictionnaire gr représentant la liste d'adjacence d'un graphe pondéré
              r un sommet du graphe
    Sorties : dictionnaire d dont les clés sont les sommets et les valeurs sont les distances
        minimales avec r
              dictionnaire parent donnant pour chaque sommet son parent dans le chemin le plus
                  court depuis r
    """
    d = {}
    etat = {}
    parent = {}
    for u in gr: # u décrit toutes les clés
        d[u] = float('inf')
        etat[u] = 'inconnu'
    d[r] = 0
    etat[r] = 'vu'
    while "vu" in etat.values():

        u = << zone à compléter >>

        etat[u] = 'traite'
        for s in gr[u]:
            v, poids = s   # s est de la forme (sommet, poids)
            if etat[v] != 'traite':

                << zone à compléter >>

    return (d, parent)
```

**Q4 –** On considère le graphe ci-dessous dont toutes les arêtes sont de poids 1 :




et l'on suppose sa liste d'adjacence représentée par un dictionnaire `gr` de sorte que l'on ait par exemple :

```
>>> gr['E']
[('C', 1), ('F', 1), ('H', 1)]
```

Poursuivre les instructions suivantes afin que la variable `chemin` contienne la liste des sommets rencontrés pour aller de A à I avec une longueur minimale (c'est-à-dire que l'on doit obtenir `['A','B','C','E','H','G','J','I']`).

```
di, pa = Dijkstra(gr, 'A')
chemin = ['I']
s = 'I'

<< à poursuivre >>
```

Nous allons maintenant nous intéresser à l'algorithme A* dont le but est également d'obtenir un chemin de longueur minimale (*i.e.* de poids minimal) entre un sommet de départ $s$ et un sommet d'arrivée $t$. On exploite pour cela une *estimation*, notée $h(u,t)$, du coût du chemin de coût minimal qu'il reste à parcourir d'un sommet $u$ à $t$ ; cette fonction $h$ est appelée une *heuristique*.
Il suffit alors essentiellement d'adapter l'algorithme de Dijkstra en ne comparant non pas les distances $d(u)$ mais en considérant $f(u) = d(u) + h(u,t)$ (*i.e.* on sélectionne les sommets voisins en cherchant celui avec $f(u)$ minimal et non $d(u)$ minimal).

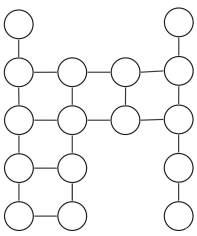
Voici une possibilité de programmation de cet algorithme.

```
1  def A_etoile(gr, s, t, h):
2      f = {} # coût total estimé
3      d = {} # distance actuelle au sommet départ
4      for x in gr:
5          d[x] = float('inf')
6      d[s] = 0
7      f[s] = h(s, t)
8      ouvert = [s] # sommets en cours d'exploration
9      ferme = []   # sommets qui ne seront plus visités
10     parent = {}
11     while ouvert != []:
12         u = choix_min(ouvert, f)
13         ouvert.remove(u)  # on enlève u de la liste ouvert
14         if u == t:
15             return (d[t], parent)
16         else:
17             ferme.append(u)
18             for cp in gr[u]:
19                 v = cp[0]
20                 poids = cp[1]
21                 if v not in ferme:
22                     d_cand = d[u] + poids
23                     if d_cand < d[v]:
24                         d[v] = d_cand
25                         f[v] = d[v] + h(v, t)
26                         parent[v] = u
27                     if v not in ouvert:
28                         ouvert.append(v)
```

**Q5 –** Écrire la fonction `choix_min` apparaissant ligne 12 dont la spécification est :

```
def choix_min(a, val):
    """
    Entrée : - liste a d'objets tous différents et totalement ordonnés
             - dictionnaire val qui a pour clés les objets
                    et pour valeurs un nombre associé à chaque objet
    Sortie : l'objet de a le plus petit parmi ceux de valeur minimale
    """
```

On s'intéresse maintenant à des graphes de la forme suivante :

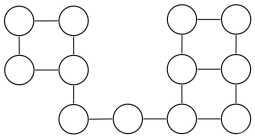


où les sommets sont alignés sur un quadrillage et reliés par des arêtes de longueur 1 verticales et horizontales (avec éventuellement des sommets «manquants» ce qui permet de modéliser des déplacements sur un quadrillage avec des obstacles à contourner).
Les sommets seront nommés par leurs coordonnées (lignes, colonne). Par exemple le sommet en haut à gauche est le sommet `(0,0)` et le sommet en bas à droite est le sommet `(4,3)`.

**Q6 –** Écrire une fonction `quadri(p,q,L)` où `p` et `q` sont des entiers strictement positifs et `L` est une liste de couples de coordonnées et qui renvoie la liste d'adjacence du graphe du type précédent avec `p` lignes et `q` colonnes, sans les sommets désignés par `L`, avec des arêtes (de poids 1) entre les sommets.
Par exemple :

```
quadri(3, 5, [(2,0), (0,2), (1,2)])
```

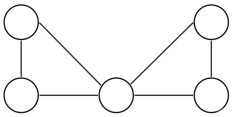
correspond au graphe suivant :



**Q7 –** Écrire deux fonctions susceptibles de constituer une heuristique :

- `euclidien(u, v)` qui calcule la distance euclidienne (à «vol d'oiseau») entre les sommets `u` et `v`,
- et `Manhattan(u, v)` qui calcule la distance de Manhattan entre les sommets `u` et `v` (c'est-à-dire en ne se déplaçant que verticalement et horizontalement, comme s'il y n'y avait pas de «sommet manquant» dans le quadrillage).

**Q8 –** Considérons maintenant un type de graphes dont les sommets sont disposés de la même façon mais avec également les arêtes en diagonale. Par exemple, le graphe suivant :



correspond à la liste d'adjacence :

```
g = {(0,0): [((1,0), 1), ((1,1), sqrt(2))],      (1,0): [((0,0), 1), ((1,1), 1)],
     (1,1): [((0,0), sqrt(2)), ((1,0), 1), ((0,2), sqrt(2)), ((1,2), 1)],
     (0,2): [((1,1), sqrt(2)), ((1,2), 1)],      (1,2): [((0,2), 1), ((1,1), 1)]}
```

Que peut-on dire des deux heuristiques vues ci-dessus dans le cas de ces graphes ?

**Q1 –**

| Étape | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 1 | × | (2, A) | (4, A) | (7, A) | $\infty$ |
| 2 | × | × | (3, B) | (7, A) | $\infty$ |
| 3 | × | × | × | (7, A) | (8, C) |
| 4 | × | × | × | × | (8, C) |

**Q2 –**

```
def vu_min(etat, d):
    L = []
    d_min = float('+inf')
    for u in etat:
        if etat[u] == 'vu':
            if d[u] < d_min:
                L = [u]
                d_min = d[u]
            elif d[u] == d_min:
                L.append(u)
    L.sort()
    return L[0]
```

**Q3 –**

```
def Dijkstra(gr, r):
    d = {}
    etat = {}
    parent = {}
    for u in gr:
        d[u] = float('inf')
        etat[u] = 'inconnu'
    d[r] = 0
    etat[r] = 'vu'
    while "vu" in etat.values():
        u = vu_min(etat, d)
        etat[u] = 'traite'
        for s in gr[u]:
            v, poids = s
            if etat[v] != 'traite':
                d_test = d[u] + poids
                if d_test < d[v]:
                    d[v] = d_test
                    parent[v] = u
                    etat[v] = 'vu'
    return (d, parent)
```

**Q4 –**

```
di, pa = Dijkstra(gr, 'A')
chemin = ['I']
s = 'I'
while s != 'A':
    s = pa[s]
    chemin.append(s)
chemin.reverse()
```

**Q5 –**

```
def choix_min(a, val):
    v_min = float('+inf')
    for x in a:
        if val[x] < v_min:
            v_min = val[x]
            x_min = x
        elif val[x] == v_min and x < x_min:
            x_min = x
    return x_min
```

**Q6 –** Voici une version possible :

```
def quadri(p, q, L):
    g = {}
    for i in range(p):
        for j in range(q):
            if (i, j) not in L:
                g[(i, j)] = []
                if i>0 and (i-1, j) not in L:
                    g[(i, j)].append((i-1, j))
                if i<p-1 and (i+1, j) not in L
                    :
                    g[(i, j)].append((i+1, j))
                if j>0 and (i, j-1) not in L:
                    g[(i, j)].append((i, j-1))
                if j<q-1 and (i, j+1) not in L
                    :
                    g[(i, j)].append((i, j+1))
    return g
```

**Q7 –**

```
from math import sqrt

def euclidien(u, v):
    a, b = u
    c, d = v
    return sqrt((c - a)**2 + (d - b)**2)

def Manhattan(u, v):
    a, b = u
    c, d = v
    return abs(c - a) + abs(d - b)
```

**Q8 –** Avec la possibilité de se «déplacer en diagonale», la distance de Manhattan risque de surestimer la distance restant à parcourir et n'est donc pas une heuristique à retenir.

## Exercice 3 – programmation dynamique

On appelle sous-liste d'une liste T = $[t_0, \cdots, t_{n-1}]$ de longueur $n$ toute liste de la forme $[t_{i_0}, \cdots, t_{i_k}]$ d'indices croissants ($0 \leqslant i_0 < i_1 < \cdots < i_k < n$). Cette sous-liste est dite *croissante* lorsque ses éléments sont rangés par ordre croissant : $t_{i_0} \leqslant t_{i_1} \leqslant \cdots \leqslant t_{i_k}$. Par convention, une liste d'un seul élément est croissante.

Par exemple, la liste [5,4,22,3,45,88,5] admet les listes [5,22,45,88] et [4,22,45,88] pour sous-listes croissantes de longueur maximale.

1 ▷ On construit une nouvelle liste A = $[a_0, \cdots, a_{n-1}]$ de la manière suivante :

$$a_i = \begin{cases} 1 & \text{si : } \forall j < i\,,\ t_j > t_i\,; \\ 1 + \max\{a_j\,;\ j < i \text{ et } t_j \leqslant t_i\} & \text{sinon.} \end{cases}$$

Calculer «à la main» la liste A obtenue pour la liste donnée en exemple ci-dessus.

2 ▷ Écrire une fonction `longueurs(T)` qui renvoie la liste A correspondante.

3 ▷ Que représente un indice $i$ tel que $a_i = 1$ ? tel que $a_i = 2$ ?

En déduire l'expression, au moyen de la liste A, de la longueur maximale d'une sous-liste croissante finissant en $t_i$.

4 ▷ Comment trouver la longueur maximale d'une sous-liste croissante de `T` en utilisant la liste A ?

En déduire une fonction `longueurmax(T)` renvoyant la longueur maximale d'une sous-liste croissante de `T`.

5 ▷ Quelle est la complexité de la fonction précédente ?

6 ▷ Écrire une fonction `listemax(T)` renvoyant une sous-liste croissante de longueur maximale de `T`.

---

1 ▷ On a $a_0 = 1$.

Pour $i = 1$ : on a $5 > 4$ *i.e.* $t_1 > t_0$ donc $a_1 = 1$.

Pour $i = 2$ : $t_2 = 22$ donc on est dans le second cas, le maximum est le maximum de 1 et 1 donc $a_2 = 2$.

Pour $i = 3$ : $t_3 = 3$ donc on est dans le premier cas et $a_3 = 1$.

Pour $i = 4$ : $t_4 = 45$ donc on est dans le second cas, le maximum est le maximum de $1, 1, 2$ et 1 donc $a_4 = 3$.

Pour $i = 5$ : $t_5 = 88$ donc on est dans le second cas, le maximum est le maximum de $1, 1, 2, 1$ et 3 donc $a_5 = 4$.

Pour $i = 6$ : $t_6 = 5$ donc on est dans le second cas, le maximum est le maximum de $a_0 = 1$, $a_1 = 1$ et $a_3 = 1$ donc $a_6 = 2$.

On a finalement A = $[1, 1, 2, 1, 3, 4, 2]$.

2 ▷
```
def longueurs(T) :
    if len(T) == 0:
        return []
    A = [1]
    for i in range(len(T)-1):
        ensemble = [ A[j] for j in range(i+1) if T[j] <= T[i+1] ]
        if ensemble == []:
            A.append(1)
        else:
            A.append(1 + max(ensemble))
    return A
```

3 ▷ Puisque les $a_i$ valent au moins 1, le max est également égal au moins à 1 donc $a_i = 1$ lorsque l'on est dans le premier cas *i.e.* lorsque $t_i$ est inférieur à toutes les valeurs précédentes ; un tel indice $i$ ne peut donc être la fin d'une sous-liste croissante que si cette dernière est de longueur 1.

Si $a_i = 2$ alors on est dans le second cas est le max vaut 1 *i.e.* il y a des valeurs $t_j$ (avec $j < i$) inférieures ou égale à $t_i$ mais elles correspondent toutes à une valeur $a_j = 1$. Autrement dit $t_i$ est la fin d'une sous-liste croissante de longueur 2 mais pas plus.

De façon générale, une sous-liste croissante finissant en $t_i$ a pour longueur maximale $a_i$.

4 ▷ La longueur maximale d'une sous-liste croissante de `T` est donc le maximum des $a_i$.

```
def longueurmax(T):
    return max(longueurs(T))
```

ou en redéfinissant le maximum d'une liste :

```
def longueurmax(T):
    A = longueurs(T)
    M = A[0]
    for i in range(1, len(A)):
        if A[i] > M:
            M = A[i]
    return M
```

5 ▷ Cette dernière fonction repose sur l'utilisation de `longueur(T)` puis la recherche du maximum.

La recherche du maximum est un $O(n)$ où $n$ est le nombre d'éléments de `T`.

La fonction `longueur` dépend d'une boucle avec $n-1$ itérations. Lors de l'itération $i$, il y a $O(i)$ opérations pour construire la liste `ensemble` puis $O(i)$ pour déterminer son maximum, on ne compte pas le `append`. Finalement, `longueur(T)` a une complexité en $O(n^2)$.

Il s'ensuit que la complexité de `longueurmax` est en $O(n^2)$.

6 ▷ L'idée est de repérer l'indice $i$ correspondant au $a_i$ maximal l'indice $j < i$ avec $a_j = a_i - 1$, etc.

```
def listemax(T) :
    A = longueurs(T)
    amax = max(A)
    indices = []
    r = len(T)-1
    for nb in range(amax, 0, -1) :
        while A[r] != nb :
            r -= 1
        indices.append(r)
    return [ T[i] for i in indices[::-1] ]
```