Interrogation d'informatique n°8. *Barème sur 24.5 pts*

Exercice A. Bases de données SQL (5.5 points)

On rappelle un exemple typique de requête SQL :

```
SELECT DISTINCT attribut1, MAX(attribut2) AS maximum
FROM table1 AS t1 JOIN table2 AS t2 ON t1.attribut3 = t2.attribut3
WHERE ...
GROUP BY ...  HAVING ...
ORDER BY DESC ...  LIMIT 3 OFFSET 2 ...
```

Opérations ensemblistes : `UNION`, `INTERSECT`, `EXCEPT` (différence)

On considère un réseau social simplifié :

Individus

id	*nom*	*prenom*
1	Mychkine	Léon
2	Filippovna	Nastassia
⋮	⋮	⋮

Liens

id1	*id2*
1	2
2	1
⋮	⋮

Lieux

id	*ville*
1	Moscou
2	Saint-Pétersbourg
⋮	⋮

La table *Individus* répertorie les individus et contient les colonnes :

– *id* (clé primaire), un entier identifiant chaque individu ;

– *nom* et *prenom* : une chaîne de caractères donnant le nom de famille de l'individu.

La table *Liens* répertorie les liens d'amitiés entre individus et contient les colonnes :

– *id1* (clé étrangère) entier identifiant le premier individu du lien d'amitié

– *id2* (clé étrangère) entier identifiant le second individu du lien d'amitié .

Noter que pour tout couple (x, y) dans la table *Liens*, le couple (y, x) est également présent dans la table.

La table *Lieux* répertorie les villes de résidence des individus (définis par la clé étrangère *id*).

1) [0.75 pt] Écrire une requête SQL qui renvoie les identifiants des amis résidant à Moscou de l'individu dont l'identifiant est 18.

2) [0.75 pt] Écrire une requête SQL qui renvoie les *nom et prenom* des amis de l'individu d'identifiant 18, classés par ordre alphabétique de leur *nom*.

3) [1 pt] Écrire une requête SQL qui renvoie les identifiants des amis de l'individu d'identifiant 18 qui ont la même ville de résidence que lui. *Conseil* : utiliser un SELECT imbriqué.

4) [1 pt] Écrire une requête SQL qui renvoie l'identifiant de l'individu possédant le plus d'amis.

On supposera qu'il n'y a pas de cas d'égalité.

5) [1 pt] Écrire une requête SQL qui renvoie sans doublon les identifiants des individus qui sont amis avec au moins un ami de l'individu d'identifiant 18.

6) [0.75 pt] Écrire une requête SQL renvoyant la liste des *nom* et *prenom* des amis communs aux individus d'identifiants 18 et 21. *Remarque* : On peut (ou non) utiliser des opérations ensemblistes.

Exercice B. Partitions équilibrées d'entiers (6.75 points)

On considère un ensemble E de n **entiers naturels**, codé en PYTHON par la liste de ses éléments.

On cherche à partitionner E en deux parties A et B de sorte à minimiser $\Delta = \left|\sum_{x\in A} x - \sum_{x\in B} x\right|$.

On note $s = \sum_{x\in E} x$ la somme de tous les éléments de E.

On peut reformuler le problème de la façon suivante :

On cherche $A \subset E$ telle que $\sum_{x\in A} x$ soit inférieur ou égal à $\frac{s}{2}$ et le plus proche possible de $\frac{s}{2}$.

La valeur minimale recherchée est alors $\Delta = s - 2j$.

Pour $0 \leq k \leq n-1$, on note $E[k]$ l'élément de E d'indice k.

Pour $0 \leq k \leq n$, on note $E[0:k]$ désigne la liste des k premiers éléments de la liste E.

On dit qu'**un entier j est une somme extraite de** $E[0:k]$ ssi il existe une partie $A \subset E[0:k]$ telle que

$$\sum_{x\in A} x = j$$

1) [0.25 pt] Écrire une fonction `somme(E)` qui renvoie la somme s de tous les éléments de E.

Par exemple, `somme([1,2,4,6])` renvoie 13

2) Méthode gloutonne

On lit les éléments x de E dans l'ordre de la liste :

Pour tout élément x, on ajoute x à la liste A lorsque la propriété $\sum_{x\in A} x \leq \frac{1}{2}s$ est conservée.

[1 pt] Écrire la fonction `glouton(E)` qui renvoie la liste A obtenue par cet algorithme.

3) Programmation dynamique de haut en bas

Pour tout $0 \leq k \leq n$ et $j \in \mathbb{N}$, on note $M(k,j)$ le plus grand entier $\leq j$ qui est une somme extraite de $E[0:k]$.

En particulier, $M(0,j) = 0$ pour tout entier $j \in \mathbb{N}$.

a) [1 pt] Pour $1 \leq k \leq n$, exprimer $M(k,j)$ en fonction de $M(k-1,j)$ et de $M(k-1, j-E[k-1])$.

On distinguera les cas $j \geq E[k-1]$ et $j < E[k-1]$.

b) [2 pts] En utilisant d'une part un dictionnaire dont les clés sont des couples (k,j) et d'autre part une fonction auxiliaire récursive, écrire une fonction `equilibre(E)` qui renvoie la valeur de Δ.

c) [0.5 pt] Donner la complexité de `equilibre`.

4) Programmation dynamique de bas en haut

On note $\boxed{m = s//2}$ le quotient de la division euclidienne de s par 2.

Pour tout $0 \leq k \leq n$, on note P_k le tableau de longueur $(m+1)$ défini par

$$\text{pour tout } j \in [\![0, m]\!] \ , \ \ P_k[j] = \begin{cases} \texttt{True} \text{ si } j \text{ est une somme extraite de } E[0:k] \\ \texttt{False} \text{ sinon} \end{cases}$$

a) [2 pts] Écrire une fonction `sommes(E)` qui renvoie le tableau P_n selon le principe suivant :

- On utilise un **unique** tableau `P` de longueur $(m+1)$ au cours de l'algorithme.
En particulier, le tableau P_0 (valeur initiale de `P`) est le tableau de longueur $(m+1)$ dont tous les éléments valent `False` à l'exception de l'élément d'indice 0 qui vaut `True` (en effet, une somme vide vaut 0).

- Puis on utilise une boucle `for` sur k de sorte qu'à la k-ième étape, le tableau `P` soit le tableau P_k.

b) [0.25 pt] Donner sans justification la complexité de `sommes(E)` en fonction de n et s (supposés non nuls).

c) [0.25 pt] En déduire une fonction `equilibreBis(E)` qui renvoie la valeur de Δ.

Exercice C. Opérations élémentaires sur les dictionnaires (4 pts)

La taille d'un dictionnaire est en $O(n)$, où n est le nombre de clés du dictionnaire.

1) Recherche des grands éléments

a) [0.75 pt] Écrire une fonction `maximum(D)` qui prend en argument un dictionnaire D non vide, renvoie la clé maximale (pour l'ordre usuel) du dictionnaire et supprime dans D l'élément de clé maximale.

On rappelle qu'on supprime l'élément de clé c dans un dictionnaire D en utilisant l'instruction : `del D[c].`
On pourra par ailleurs utiliser la liste (ici non vide) des clés `D.keys()`.

Attention : On demande un algorithme de complexité linéaire $O(n)$, où n est le nombre de clés de D, et qui effectue exactement $n-1$ comparaisons entre clés de D.

b) [0.75 pt] Écrire une fonction `maxima(D,k)` qui prend en arguments un dictionnaire de n clés et un entier $k \leq n$, et renvoie la liste des k plus grandes clés. On autorise de modifier D au cours de l'algorithme. Il est attendu ici une complexité en $O(nk)$ dans le pire cas.

2) Partitions par valeurs

On considère un dictionnaire. On peut regrouper les clés selon la valeur associée.

Par exemple, si `Dico = {a : 1 , b : 5 , c : 1 , d : 3 }`, les classes sont `[a,c]` , `[b]` , `[d]`.

Attention : On demande des algorithmes de complexité linéaire en la taille des arguments.

On pourra construire et utiliser des dictionnaires auxiliaires.

a) [1.5 pt] Écrire une fonction `classes(D)` qui prend en argument un dictionnaire `D`, et renvoie la liste des classes des clés. L'ordre des classes et l'ordre au sein de chaque classe sont arbitraires.

Par exemple, `classes(Dico)` renvoie (par exemple) la liste de listes `[[a,c],[b],[d]]`.

b) [1 pt] Écrire une fonction `clés(D1,D2)` qui prend en arguments deux dictionnaires `D1` et `D2`, et renvoie la liste des classes des clés communes à `D1` et `D2` regroupées selon le couple des valeurs de `D1` et `D2`.

Autrement dit, deux clés `x` et `y` sont dans une même classe ssi leurs valeurs respectives dans `D1` et `D2` existent et sont égales, c'est-à-dire ssi `D1[x] == D1[y]` et `D2[x] == D2[y]`.

Exercice D. Permutation de tri (2.25 points)

1) [1 pt] Écrire une procédure `tri(L,f)` qui étant donnée une liste $L = [a_0, a_1, ..., a_{n-1}]$ trie ses éléments selon le procédé **du tri par insertions** en les classant de sorte que $f(a_0) \leq f(a_1) \leq ... \leq f(a_{n-1})$.

Autrement dit, à la k-ième, l'ordre des éléments de la sous-liste $L[0 : k]$ a été modifié de sorte que $f(a_0) \leq f(a_1) \leq ... \leq f(a_{k-1})$.

2) [1.25 pt] Écrire une fonction `numeros(L)` qui étant donnée une liste d'entiers $L = [a_0, a_1, ..., a_{n-1}]$ renvoie une liste S codant une permutation de $[\![0, n-1]\!]$ telle que $a_{S(0)} \leq a_{S(1)} \leq ... \leq a_{S(n-1)}$.

Suggestion : Trier les couples `(L[k],k)` avec 1) et définir S à partir de la liste triée obtenue.

Exercice E. Recherche des k plus grands éléments en utilisant un tournoi (6 pts)

L'algorithme des k plus proches voisins nécessite de déterminer les k éléments optimaux dans la liste des distances à des configurations connues. On s'intéresse ici à optimiser cette recherche.

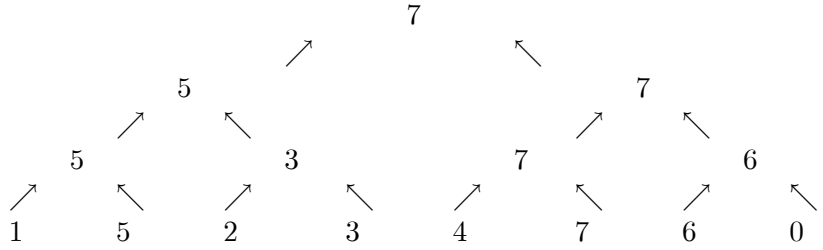
Le calcul du plus grand élément d'une liste demande $n-1$ comparaisons entre éléments de la liste.
On peut donc calculer les k plus grands éléments avec au plus kn comparaisons. On souhaite améliorer cette complexité.

1) [0.25 pt] Préciser sans justification si l'algorithme des k plus proches voisins intervient dans le contexte d'un apprentissage supervisé ou bien d'un apprentissage non-supervisé.

2) [0.25 pt] **Notion de tournoi**

Soit $p \in \mathbb{N}^*$. On associe à une liste de $\boxed{n = 2^p}$ entiers distincts $L = [x_0, x_1 ... x_{n-1}]$ un arbre binaire dont **les feuilles sont étiquetées par les entiers** x_i. Et l'étiquette de chaque sommet interne (appelé nœud) est le maximum des étiquettes de ses deux fils.

Exemple : avec $n = 8$ et $L = [x_0, x_1, ..., x_7] = [1, 5, 2, 3, 4, 7, 6, 0]$, on considère donc



Cette situation correspond à un tournoi où on regroupe les éléments par groupes de deux et chaque duel entre deux éléments aboutit à l'élimination du plus petit d'entre eux. La phase suivante consiste à l'affrontement par duels entre les vainqueurs. Le procéde se poursuit jusqu'à ce que seul l'élément maximum reste en lice.

Justifier qu'il y a exactement $n-1$ comparaisons (= duels) entre les x_i effectués lors du tournoi.

3) *Implémentation en* PYTHON. La fonction `max(x,y)` est supposée connue.

On souhaite généraliser et implémenter le procédé du tournoi aux listes **de longueur n arbitraire.**

A toute liste d'entiers $L = [x_0, x_1, ..., x_{n-1}]$ de longueur n, on associe la liste $f(L)$ définie en groupant les termes par deux, et en remplaçant chaque paire d'entiers par leur maximum.

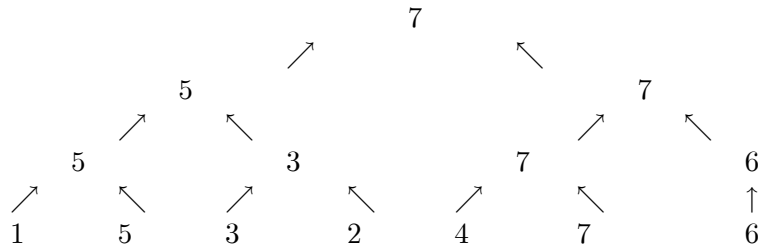
Dans le cas d'une liste de longueur impaire, le dernier terme est conservé.

Autrement dit, si $n = 2m$ est pair, $f(L) = [y_0, y_1, ..., y_{m-1}]$ où $\forall i \in \{0, 1, ..., m-1\}$, $y_i = \max(x_{2i}, x_{2i+1})$.

Et si $n = 2m+1$ est impair, $f(L) = [y_0, y_1, ..., y_{m-1}, x_{n-1}]$.

Par exemple, si $L = [1, 5, 3, 2, 4, 7, 6]$, alors $f(L) = [5, 3, 7, 6]$, car $\max(1, 5) = 1$, $\max(3, 2) = 3$ et $\max(4, 7) = 7$.

Le tournoi associé à $L = [1, 5, 3, 2, 4, 7, 6]$ peut être représentée par le schéma suivant :



a) [1 pt] Étant donnée une liste L non vide, on considère les listes obtenues par applications successives de f.
En un nombre fini d'étapes, on aboutit à la liste $[z]$, où z est la valeur maximale de L.

Écrire en PYTHON une fonction `maximum(L)` qui étant donnée une liste non vide L renvoie la valeur maximale de cette liste en utilisant le principe du tournoi décrit ci-dessus.

b) [1 pt] Démontrer que cet algorithme effectue exactement $(n-1)$ comparaisons entre éléments de la liste.

4) On se donne une liste L de longueur n entiers naturels distincts deux à deux.

On souhaite calculer de façon efficace **les k plus grands éléments** de L.

a) [1.25 pt] Écrire une fonction `versus(L)` qui étant donné la liste L renvoie le couple (z, V), où

- z est la valeur maximale de L

- V est le dictionnaire dont les clés sont les éléments de L et tel que pour toute clé x, $V[x]$ est la liste des éléments qui ont été battus par x lors du tournoi.

Par exemple, dans l'exemple ci–dessus, l'élément maximal est $z = 7$ et le dictionnaire V est

```
{1:[] ; 2:[] ; 3:[2] ; 4:[] ; 5:[1,3] ; 6:[] ; 7:[4,6,5]}
```

On demande une fonction de complexité $O(n)$ et effectuant exactement $n-1$ comparaisons entre éléments de L.

b) [0.25 pt] Pour déterminer le deuxième plus grand élément, il suffit de calculer la valeur maximale de la liste `V[z]` des éléments battus par l'élément maximal z au cours du tournoi.

Justifier brièvement qu'on peut ainsi calculer les deux plus grands éléments de L en effectuant au plus $(n - 2 + \lceil \log n \rceil)$ comparaisons entre deux éléments de L.

5) [2 pts] a) En généralisant la méthode du 4), décrire très brièvement un algorithme renvoyant la liste `Grands` des k plus grands éléments de L.

On utilisera le dictionnaire V défin dans 4) et un dictionnaire `Battus` dont les clés sont les éléments de L qui ont été battus lors du tournoi par un des plus grands éléments déjà déterminés (et les valeurs des clés sont arbitraires) et utiliser la fonction `maximum` de la question 1) a) de l'exercice C.

Il est conseillé d'initialiser par : `Battus = {z:1} ; Grands = []`.

b) Écrire une fonction `maxi(L,k)` qui renvoie la liste des k plus grands éléments de L.

c) Préciser sans justification la complexité de la fonction `maxi(L,k)` en donnant un majorant du nombre de comparaisons effectuées entre éléments de L sous la forme $n + \varphi(k, n)$, où φ est une fonction à préciser.

Compte tenu de la complexité des algorithmes de tri, pour quel ordre de grandeur de k la méthode décrite ici est-elle pertinente ?

d) On peut utiliser à la place du dictionnaire `battus` une file de priorité (implémentée par un tas) où chaque ajout d'un élément et chaque retrait du maximum demande au plus $\lceil \log m \rceil$ comparaisons, où m est la longueur du tas. Quelle est la complexité alors obtenue ?

6) *Question supplémentaire. Optimalité du calcul du maximum*

On va montrer que tout algorithme calculant le maximum effectue au moins $(n-1)$ comparaisons.

Considérons un algorithme $\mathcal{A}$ prenant en entrée n entiers, et retournant le plus grand de ces entiers. On suppose que cet algorithme exécute des comparaisons entre éléments du tableau, et que le résultat retourné ne dépend que des résultats des comparaisons effectuées.

On suppose également qu'il existe une entrée $L = [x_0, x_1 ... x_{n-1}]$ telle que $\mathcal{A}$ exécute strictement moins de $(n-1)$ comparaisons, lorsqu'il est exécuté sur l'entrée L.

a) Soit un graphe non orienté $G = (S, A)$ admettant n sommets.
On dit que G est connexe ssi pour tous sommets i et j, il existe un chemin dans G reliant i et j.

Montrer tout graphe connexe G admet au moins $(n-1)$ arêtes.

b) Construire une entrée $L' = [x'_0, x'_1 ..., x'_{n-1}]$ telle que l'algorithme ne retourne pas le plus grand élément de x'.

c) Montrer que tout algorithme calculant le maximum effectue au moins $(n-1)$ comparaisons.