**Interrogation d'informatique n°8. Corrigé**

**Exercice A**

**1)**
```
SELECT id1 FROM Liens JOIN Lieux ON id1 = id
WHERE id2 = 18 AND ville = 'Moscou'
```

On utilise ici la symétrie des relations dans la table.

**2)**
```
SELECT nom, prenom FROM Individus JOIN Liens ON id1 = id
WHERE id2 = 18
ORDER BY nom
```

**3)**
```
SELECT id1 FROM Liens JOIN Lieux ON id1 = id
WHERE id2 = 18 AND ville = (SELECT ville FROM Lieux WHERE id = 18)
```

**4)**
```
SELECT id1 FROM Liens
GROUP BY id1
ORDER BY COUNT(*) DESC
LIMIT 1
```

Autre solution :

```
SELECT id1
FROM ( SELECT id1, COUNT(*) AS nombre GROUP BY id1 )
WHERE nombre = MAX(nombre)
```

**5)**
```
SELECT DISTINCT table1.id2
FROM Liens AS table1 JOIN Liens AS table2 ON table1.id1 = table2.id1
WHERE table2.id2 = 18 ANS table1.id2 != 18
```

Autre solution :

```
SELECT id1 FROM Liens
WHERE id2 IN
     ( SELECT id1 FROM Liens WHERE id2 = 18)
```

**6)** Première solution :

```
( SELECT nom, prenom FROM Individus JOIN liens ON id1 = id
WHERE id2 = 18 )
INTERSECT
( SELECT nom, prenom FROM Individus JOIN liens ON id1 = id
WHERE id2 = 21 )
```

Seconde solution :

```
SELECT DISTINCT table1.id2
FROM liens AS table1 JOIN Liens AS table2 ON table1.id2 = table2.id2
```

```
WHERE table1.id1 = 18 AND table2.id1 = 21
```

**Exercice B**

**1)**

```
def somme(E) :
       s = 0
       for x in E : s = s + x
       return s
```

**2)**

```
def glouton(E) :
     s = somme(E) ; t = 0 ; A = []     # t vaut la somme des éléments de A
     for x in E :
          if 2*(t + x) <= s :
                A.append(x) ; t = t + x
     return A
```

**3)** a) En envisageant les sommes extraites contenant ou non $E[k-1]$, on obtient :

$$M(k+1,j) = \begin{cases} \max(\ M(k-1,j)\ ,\ M(k-1,j-E[k-1]) + E[k-1]\ ) \text{ si } j \geq E[k-1] \\ M(k-1,j) \text{ si } j < E[k-1] \end{cases}$$

b)
```
def equilibre(E) :
     def aux(k,j) :     # définit dico[(k,j)] s'il ne l'est pas déjà
         if (k,j) in dico :  return None
         if k == 0 :  dico[(k,j)] = 0
         aux(k-1,j)
         if j >= E[k-1] :
             aux(k-1,j-E[k-1])
             dico[(k,j)] = max(dico[(k-1,j),dico[(k-1,j-E[k-1])+E[k-1]]
         else :
             dico[(k,j)] = dico[(k-1,j)]
     n = len(E) ; dico = {} ; s = somme(E)
     aux(n,s//2) ; return s - 2*dico[(n,s//2)]
```

c) Chaque valeur de `dico[(k,j)]`, avec $0 \leq k \leq n$ et $0 \leq j \leq s//2$ est construite au plus une fois.
D'où une complexité (dans le pire cas) en $O(ns)$.

**4)** a)

```
def sommes(E) :
    s = somme(E) ; m = s//2 ;
    P = [False]*(m+1) ; P[0] = True
```

```
    for x in A :
         Q = P.copy()    # Q mémorise l'ancienne valeur de P
         for i in range(m+1-x) :
              if Q[i] :  P[i+x] = True    # valeurs de P modifiées
    return P
```

Variante : On utilise une boucle `for i ...` décroissante afin d'éviter d'avoir à faire une copie.

b)

```
def equilibreBis(E) :
     P = sommes(E) ; s = len(P)-1
     j = s//2
     while not P[j] :  j = j-1
     return s-2*j
```

c) La construction de $P_n$ demande $O(ns)$ opérations, puisque le passage de $P_k$ à $P_{k+1}$ demande $O(s)$ opérations.

**Exercice C**

**1)** a)
```
def maximum(D) :
       m = 0
       for x in D :
            if D[x] > m :
               z = x ; m = D[z]
       del D[z)
       return z
```

b)
```
def maxima(D,k) :
       M = []
       for _ in range(k) :  M.append(maximum(D))
       return M
```

**2)** a) On crée un dictionnaire auxiliaire `Valeurs` dont les clés sont les valeurs $v$ de $D$ et `Valeurs[v]` est la liste des clés de $D$ dont la valeur est $v$.

```
def classes(D) :
     Valeurs = {}
     for v in D.keys() :  Valeurs[v] = []
     for x in D : Valeurs[D[x]].append(x)
     return [ Valeurs[v] for v in Valeurs]
```

b) On crée un dictionnaire auxiliaire `Couples` dont les clés sont les clés communes à `D1` et `D2` et dont les valeurs sont les couples des valeurs associées, puis on utilise 1).

```
def classes(D1,D2) :
```

```
    Couples = {}
    for x in D1 :
        if x in D2 :
            Couples[x] = (D1[x],D2[x])
    return classes(D)
```

**Exercice D. Permutation de tri**

**1)**

```
def tri(L,f) :
    n = len(L)
    for i in range(1,n) :
        k = i
        while k>0 and f(L[k-1])<f(L[k]) :
            L[k-1],L[k] = L[k],L[k-1]
    return L
```

**2)** On trie les couples `(L[k],k)` selon la valeur des `L[k]`

On récupère la liste $S$ en considérant les valeurs des $k$ dans l'ordre ainsi obtenu.

```
def f(couple) :  return couple[0]
def numeros(L) :
    n = len(L)
    M = [(L[k],k) for k in range(n)] ; tri(M,f)
    return [ M[k][1] for k in range(n) ]
```

Autre méthode : On manipule le tableau des `L[S[k]]`.

```
def numeros(L) :
    n = len(L) ; S = [k for k in range(n)]
    for i in range(n) :
        k = i
        for j in range(i+1,n) :
            if L[S[j]]<L[S[k]] :  k = j
        S[i],S[k] = S[k],S[i]
    return S
```

**Exercice E**

**1)** Il s'agit d'une méthode utilisée dans le cadre des apprentissages supervisés (et les plus proches voisins sont des données de référence).

**2)** a) Pour passer d'un niveau au niveau précédent de l'arbre, on utilise un nombre de comparaisons égal à la moitié du nombre d'éléments.

Donc le nombre de comparaisons nécessaires vaut $2^{p-1} + 2^{p-2} + ... + 1 = 2^p - 1 = n - 1$.

b) Lors des combats de la première phase entre les $n = 2^p$ éléments, on récupère d'une part la liste des $2^{p-1}$ vainqueurs et d'autre part la liste des $2^{p-1}$ vaincus.

On peut alors calculer le maximum de la première en $(2^{p-1} - 1)$ comparaisons.

Et de façon analogue, on peut alors calculer le minimum de la seconde en $(2^{p-1} - 1)$ comparaisons.

On obtient respectivement le maximum et le minimum de la liste initiale.

Et le nombre total de comparaisons est bien $2^{p-1} + 2(2^{p-1} - 1) = \frac{3n}{2} - 2$ comparaisons.

**3)** On peut utiliser une boucle while ou une fonction récursive.

Version itérative :

```
def maximum(L) :
     while len(L)>1 :
         M = [] ;  n = len(L)
         for i in range(n//2)
               M.append(max(L[2*i],L[2*i+1])
          if n%2 == 1 :  M.append(L[n-1}
         L = M
     return L[0]
```

Version récursive :

```
def maximum(L) :
     if len(L)>1 :  return L[0]
     M = [] ;  n = len(L)
     for i in range(n//2)
          M.append(max(L[2*i],L[2*i+1])
     if n%2 == 1 :  M.append(L[n-1}
     return maximum(M)
```

n montre par cet argument que le nombre d'arêtes est $n-1$, lorsque $n$ est le nombre de feuilles.

b) - *Première solution* : On raisonne par récurrence forte sur $n$, la longueur de $L$.

La propriété est immédiate pour $n = 1$.

Supposons la propriété vraie pour tous les entiers compris entre 1 et $n - 1$.

On construit $M = f(L)$ en utilisant exactement $p = \left\lfloor \frac{n}{2} \right\rfloor$ comparaisons.

D'autre part, la longueur $m$ de $f(L)$ vaut $m = \left\lceil \frac{n}{2} \right\rceil$.

Par hypothèse de récurrence, le maximum de $f(L)$ se calcule en $m - 1$ comparaisons.

Donc le maximum de $L$ se calcule en $(m - 1) + p = (n - 1)$, car $\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n$.

*Variante* : On distingue les cas $n$ pair et $n$ impair.

Si $n$ pair, $m = p = \frac{1}{2}n$ et si $n$ est impair, $m = \frac{1}{2}(n-1)$ et $p = \frac{1}{2}(n+1)$. Dans les deux cas, $p + (m - 1) = n - 1$.

- *Seconde solution* (*plus élégante*) : On se sert du principe des tournois : à chaque comparaison, on supprime un élément dans la liste, jusqu'à ce qu'il reste un seul élément. D'où $n-1$ comparaisons.

*Remarque* : Le schéma de l'algorithme peut se représenter par un arbre binaire où les feuilles sont les éléments de la liste, et chaque nœud est étiqueté par le maximum de ses deux fils. On montre par cet argument que le nombre d'arêtes est $n-1$, lorsque $n$ est le nombre de feuilles.

**4)** a)
```
def versus(L) :
    V = {}
    for x in L : V[x] = []
    while len(L)>1 :
        n = len(L) ; M = []
        for k in range(0,n,2)
            x = L[2k] ; y = L[2k+1]
            if x>y :
                M.append(x) ; V[x].append(y)
            else :
                M.append(y) ; V[y].append(x)
        if n%2 == 1 :  M.append(L[n-1])
        L = M
    return L[0],V
```

b) Il y a $(n-1)$ comparaisons. Par ailleurs, la liste $V[z]$ des battus par le vainqueur du tournoi contient au plus $\lceil \log n \rceil$ éléments (en effet, si $n \leq 2^p$, il y a au plus $p$ duels avec $z$).
Pour calculer le maximum d'une liste de longueur $p$, il faut $p-1$ comparaisons.
D'où le calcul du deuxième plus grand en effectuant au plus $(n-2+\lceil \log n \rceil)$ comparaisons.

**5)** a)
```
def maxi(L,k) :
    z,V = versus(L) ; Battus = {z:1} ; Grands = []
    for _ in range(k) :
         z = maximum(Battus) ; Grands.append(z)
         for x in V[z] :
              if not(x in Battus) :  battus[x] = 1
    return Grands
```

b) La liste des battus est de longueur $m \leq k \lceil \log n \rceil$.
Il y a $k$ recherches de maximum dans cette liste. D'où au plus $n + k \times k \lceil \log n \rceil$ comparaisons.
Le tri d'une liste demande au plus $n \lceil \log n \rceil$ comparaisons. Donc la méthode proposée ici n'est intéressante que si $k^2 = \mathrm{o}(n)$, c'est-à-dire $k = \mathrm{o}(\sqrt{n})$.

c) Le tas des battus est de longueur $m$, avec $m \leq k \lceil \log n \rceil$
Il y a $m$ ajouts au tas et $k$ suppressions, donc au plus $m \lceil \log m \rceil$ comparaisons sont nécessaires.
D'où au plus $n + m \lceil \log m \rceil$ comparaisons. On a $m \lceil \log m \rceil = O(k \log n(\log k + \log\log n))$.

Cette gestion du tas est donc profitable dès que $k >> \log\log n$.

**6) Optimalité du calcul du maximum**

On se propose de montrer que le tournoi est un algorithme optimal en nombre de 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 des éléments du tableau, et que le résultat retourné ne dépend que de l'ensemble des résultats des comparaisons effectuées.
On suppose également qu'il existe une entrée $x = \{ x_0, x_1...x_{n-1}\}$ telle que $\mathcal{A}$ exécute strictement moins de $(n-1)$ comparaisons, lorsqu'il est exécuté sur l'entrée $x$.

a) On raisonne par récurrence sur $n = \operatorname{card} G$. La propriété est mmédiate pour $n = 1$.

Supposons lapropriété vraie pour tout graphe connexe admettant au plus $(n-1)$ sommets.
Considérons une arête $a = (i, j)$ de $G$. Notons $G' = (S, A \setminus \{a\})$.
Le graphe $G'$ admet au plus deux composantes connexes.
En effet, soit $s$ un sommet de $G$. Il existe dans $G$ un chemin réduit reliant $s$ à $i$
Si ce chemin ne passe pas par l'arête $a$, alors ce chemin relie $s$ à $i$ dans $G'$.aussi.
Sinon, ce chemin passe par $j$ avant d'atteindre $i$, et donc ce chemin relie $s$ à $j$ dans $G'$.
On en déduit que tout sommet appartient à la composante connexe de $i$ ou à la composante connexe de $j$.

Si elles sont distinctes, de cardinal $k$ et $n-k$, on peut leur appliquer l'hypothèse de récurrence.
Donc $G'$ admet au moins $(k-1)+(n-k-1)$ arêtes, c'est-à-dire $n-2$ arêtes.
Donc $G$ admet au moins $n-1$ arêtes.

Si $G'$ est connexe, on peut itérer le procédé et supprimer des arêtes une à une jusqu'à déconnecter le graphe (en deux composante connexes) et conclure de façon analogue.

b) On considère le graphe $G = (S, A)$, où $S = [\![0, n-1]\!]$ et où $A$ est l'ensemble des paires $\{i, j\}$ tels que dans l'exécution de l'algorithme $\mathcal{A}$, les entiers $x_i$ et $x_j$ ont été comparés au moins une fois.

Par a), $G$ n'est pas connexe. L'idée est d'ajouter un même entier aux $x_i$ appartenant à une même composante connexe. Cette opération ne modifie pas le résultat des comparaisons effectuées par l'algorithme $\mathcal{A}$, et donc ne change pas l'indice de l'élément renvoyé.
Or, en ajoutant un entier assez grand aux éléments d'une composante connexe qui n'est pas celle du maximum des $x_i$, on obtient une nouvelle entrée $L'$ pour laquelle le maximum n'est pas la valeur renvoyée par l'algorithme.

c) On a montré que tout algorithme $\mathcal{A}$ effectuant moins de $(n-1)$ comparaisons ne peut renvoyer le maximum pour chaque entrée possible. On conclut par contraposition.