Concours CPGE EPITA-IPSA-ESME 2023
Option Sciences du Numérique
Filières MP/PC/PSI/PT/TSI
Corrigé

I. SQL

Question 1. Une clé primaire est un ensemble d'attributs permettant d'identifier un enregistrement de manière unique. Une clé primaire peut donc contenir plusieurs attributs. Pour la table `appartient`, le couple `(rid, num)` forme une clé primaire, et il n'est pas possible d'enlever un attribut.

Question 2. 1. Nombre d'entrées de la table `rando` :

```
SELECT COUNT(*) FROM rando
```

2. Abscisses et ordonnées minimales et maximales des points de passage de la table `pp` :

```
SELECT MIN(x), MAX(x), MIN(y), MAX(y) FROM pp
```

3. Identifiants des points de passage de la randonnée « Le mont chauve » :

```
SELECT pid
FROM rando JOIN appartient
ON id = rid
WHERE nom = 'Le mont chauve'
```

4. Pour chaque identifiant de randonnée le nombre de points de passage, seulement les randonnées ayant au moins 10 points de passage :

```
SELECT rid, COUNT(*)
FROM appartient
GROUP BY rid
HAVING COUNT(*) >= 10
```

5. Couples d'identifiants de randonnées différentes qui partagent au moins un point de passage :

```
SELECT DISTINCT a1.rid, a2.rid
FROM appartient a1 JOIN appartient a2
ON a1.pid = a2.pid
WHERE a1.rid <> a2.rid
```

II. Calculer le temps de trajet

Question 3.

```
f=open('rando.csv','r')
Lrando = []
T=f.readlines()
for i in range(2,len(T)):
    a,b,c=T[i].split(';')
    Lrando.append((int(a), int(b), int(c)))
```

Question 4. Par exemple :

```
def calcule_d_pente(L):
    n=len(L)
    T=[]
    for i in range(n-1):
        xi,yi,zi=L[i]
        xj,yj,zj=L[j]
        dist = sqrt((xi-xj)**2+(yi-yj)**2+(zi-zj)**2)
        dist_h = sqrt((xi-xj)**2+(yi-yj)**2)
```

```
        pente = (zj-zi)/disth
        T.append((dist, pente))
    return T
```

Question 5. On maintient l'invariant de boucle `Pente[g]` $\leq$ `p` $<$ `Pente[d]`. Lorsque $d = g + 1$, l'indice g convient.

```
def indice_pente(p):
    n=len(Pente)
    g, d = 0, n-1
    while d-g>1:
        m=(g+d)//2
        if Pente[m]<=p:
            g=m
        else:
            d=m
    return g
```

Question 6. On peut par exemple procéder en supposant une relation affine de la vitesse sur l'intervalle $[p_i, p_{i+1}[$. On obtient alors la formule :

$$v = \frac{v_{i+1} - v_i}{p_{i+1} - p_i}(p - p_i) + v_i$$

Question 7.

```
def temps_rando(L):
    T=calcule_d_pente(L)
    t = 0
    for d,p in T:
        i = indice_pente(p)
        v = (Vitesse[i+1]-Vitesse[i])/(Pente[i+1]-Pente[i])*(p-Pente[i])+Vitesse[i]
        t+=d*v
    return t
```

III. Faire son sac...

III. 1. Un algorithme glouton pour le remplissage du sac

Question 8.

```
def tri_selection(L):
    n=len(L)
    for i in range(n-1):
        imin = i
        for j in range(i+1,n):
            if L[j]<L[imin]:
                imin = j
        L[i], L[imin] = L[imin], L[i]
```

Question 9. La complexité est $O(n^2)$.

Question 10. Il suffit de changer la ligne `if L[j]<L[imin]:` par `if L[j][2]/L[j][1]>L[imin][2]/L[imin][1]:`.

On suppose dans la suite de cette sous-section que la liste d'objets est triée par valeurs décroissantes de ce ratio.

Question 11.

```
def remplissage_glouton(L, C):
    T=[]
    p=0
    for x in L:
        if x[1]+p<=C:
            T.append(x[0])
            p+=x[1]
    return T
```

Question 12. La complexité est simplement $O(n)$.

Question 13. Il suffit de reprendre l'exemple du sujet : la liste triée par valeurs décroissantes du ratio score/poids est `[('telephone', 100, 50), ('eau', 5000, 200), ('barbecue', 20000, 80)]`. Avec une capacité de $C = 5000$, l'algorithme choisit de prendre uniquement le téléphone pour un score de 50, alors que l'eau seule rapportait un score 200.

III. 2. Une recherche exhaustive

Question 14.

```
def toutes_listes(n):
    if n==0:
        return [[]]
    else:
        L=toutes_listes(n-1)
        return [x+[0] for x in L]+[x+[1] for x in L]
```

Question 15.

```
def poids_score(L, D):
    p, s = 0, 0
    for i in range(len(D)):
        if D[i]==1:
            p+=L[i][1]
            s+=L[i][2]
    return p,s
```

Question 16.

```
def remplissage_optimal(L, C):
    n=len(L)
    smax, Dmax = -1, None
    for D in toutes_listes(n):
        p, s=poids_score(L, D)
        if p<=C and s>smax:
            smax = s
            Dmax = D
    return [L[i][0] for i in range(n) if Dmax[i]==1]
```

Question 17. L'algorithme renvoie le remplissage optimal. Dans l'exemple choisi, simplement `['eau']`.

Question 18. Il y a 2^n listes à tester. Pour chacune, le calcul de `poids_score` se fait en $O(n)$, la complexité est donc $O(n2^n)$.

III. 3. Une programmation dynamique

Question 19.

1. $M_{c,0} = 0$ pour $0 \leq c \leq C$.
2. $M_{0,i}$ pour $0 \leq i \leq n$.
3. Si $p_{i-1} > c$, on ne peut pas prendre O_{i-1} sans dépasser le poids c, donc $M_{c,i} = M_{c,i-1}$. Si $p_{i-1} \leq c$, le score maximal que l'on peut obtenir en prenant l'objet O_{i-1} est $s_{i-1} + M_{c-p_{i-1},i-1}$, et sans prendre l'objet $M_{c,i-1}$. $M_{c,i}$ est donc le maximum de ces deux quantités.

Question 20.

```
def scores_optimaux(L, C):
    n=len(L)
    M=init(n, C)
    for c in range(1, C+1):
        for i in range(1, n+1):
            if L[i-1][1]<=c:
                M[c][i]=max(M[c][i-1], L[i-1][2]+M[c-L[i-1][1]][i-1])
            else:
                M[c][i]=M[c][i-1]
    return M
```

Question 21. Il suffit de remonter de la case d'indice $(c, i) = (C, n)$ jusqu'à la case $(0, 0)$. Si $M_{c,i} = M_{c,i-1}$, alors on ne prend pas l'objet O_{i-1} (et on passe à la case $(c, i-1)$), sinon on le prend et on passe à la case $(c - p_{i-1}, i-1)$.

```
def remplissage_optimal2(L,C):
    n=len(L)
    M=scores_optimaux(L, C)
    obj = []
    c, i = C,n
    while i>0:
        if M[c][i]==M[c][i-1]:
            i-=1
        else:
            obj.append(L[i-1][0])
            c-=L[i-1][1]
            i-=1
    return obj
```

Question 22. La complexité est en $O(nC)$ à cause de l'appel à `scores_optimaux`. Le reste de la fonction est en $O(n+C)$, ce qui est négligeable.

Question 23. Il faut préférer l'approche de la section III.2 lorsque $C \geq 2^n$, celle de la section III.3 si $C < 2^n$.

III. 4. Variante fractionnaire

Question 24.

1. Soit $\varepsilon > 0$ qu'on fixera plus tard. Si $x_i < 1$ et $x_j > 0$, on peut changer les choix effectués en prenant une fraction $x_i + \varepsilon$ de l'objet numéro i (ce qui rajoute un poids εp_i) et une fraction $x_j - \varepsilon\frac{p_i}{p_j}$ de l'objet j (ce qui enlève un poids $\varepsilon\frac{p_i}{p_j}p_j = \varepsilon p_i$). Le poids est donc inchangé, mais le score est augmenté de $\varepsilon s_i - \varepsilon\frac{p_i}{p_j}s_j = p_i(\varepsilon r_i - \varepsilon r_j) > 0$. Ceci est possible pourvu que $x_i + \varepsilon \leq 1$ et $x_j - \varepsilon\frac{p_i}{p_j} \geq 0$, prendre $\varepsilon = \min(1 - x_i, x_j\frac{p_j}{p_i})$ convient.
2. L'approche gloutonne est donc optimale, car elle consiste à prendre au maximum les objets de ratio les plus élevés.