Corrigé de Mines-Ponts Informatique commune 2025

Q1.

```
SELECT idC, val/taille AS ratio FROM CONTENEURS
WHERE portDepC = ''Marseille'' AND portDestC = ''Barcelone''
AND dateDisp < 2025-01-01
ORDER BY ratio DESC;
```

Q2.

```
SELECT idN, COUNT(idC)
FROM CONTENEURS
GROUP BY idN;
```

Remarque : une jointure avec la table `NAVIRES` est inutile ici. Par contre cette requête ne permet pas d'obtenir les identifiants des navires qui n'ont aucun conteneur. Une solution avec requête imbriquée comme argument de la fonction agrégative `COUNT` :

```
SELECT idN, COUNT(SELECT idC FROM CONTENEURS AS C
WHERE C.idN = N.idN)
FROM NAVIRES AS N;
```

Q3.

```
def profit(obj, S):
    res = 0
    for i in range(len(S)):
        res += obj[i][1] * S[i]
    return res
```

Q4.

```
def contrainte(obj, S, b):
    c = 0
    for i in range(len(S)):
        c += obj[i][0] * S[i]
    return c <= b
```

Q5. feuille b : `[1, 1, 0]` et feuille c : `[1, 0, 1]`.

Q6. Pour chaque objet il y a 2 possibilités, donc :

l'arbre binaire pour n objets a 2^n feuilles.

Pour chaque feuille il faut faire appel aux fonctions `profit` et `contrainte` chacune de complexité linéaire (une boucle de n tours qui ne contient que des opérations de complexité bornée) et il y a 2^n feuilles, le calcul de maximum se faisant pour chaque feuille en temps constant on en déduit que :

un algorithme de type "force brute" serait de complexité en $O(n \times 2^n)$.

Q7. Pour `obj = [(2, 3), (1, 4), (4, 4)]`, la liste `Lqi` après initialisation (lignes 4 à 6) est `[1.5, 4, 1]`. Le premier objet selectionné est alors celui d'indice `1` car il a le plus grand rapport, ce qui donne $b = 5 - 1 = 4$, puis l'objet d'incide `0` est selectionné (rapport `1.5`), ce qui donne $b = 4 - 2 = 2$. On ne peut alors plus prendre l'objet d'indice `2` (ressource $4 > 2$).
Donc,

La liste construite par l'algorithme glouton est alors : `S = [1, 1, 0].`

Q8.

```
 l5 : Lqi.append(obj[i][1]/obj[i][0])

l12 : Li[j] = Li[j - 1]

l15 : Li[j] = i
```

Q9. La méthode de tri utilisée est le tri par insertion.

- Le meilleur des cas correspond à une liste triée par ordre décroissant, il n'y a alors aucun passage dans la boucle **`while`**, il y a $n-1$ tours de boucle **`for`** de complexité bornée. Donc,

  dans le meilleur des cas la complexité est linéaire : $O(n)$.

- Le pire des cas correspond à une liste triée dans l'ordre croissant, il y a alors `i` tours de boucle while pour le tour `i` de la boucle **`for`** (la condition `Lqi[j - 1] < x` est toujours vraie) et la complexité d'un tour est bornée et

$$\sum_{i=1}^{n-1}\left(\sum_{j=1}^{i} 1\right) = O(n^2).$$

  Donc :

  la complexité dans le pire des cas est $O(n^2)$.

Q10.

```
l4 : while b > 0 and j < len(Li)
l5 : if obj[Li[j]][0] <= b:
l6 : S[Li[j]] = 1
l7 : b -= obj[[Li[j]][0]
```

Q11. On ne peut pas prendre les 3 objets à cause de la contrainte de ressource, mais on peut prendre les objets 1 et 2 qui sont ceux de plus grande valeur. On obtient alors

la solution optimale : `S = [0, 1, 1]` de profit 8.

L'approche gloutonne ne permet d'obtenir qu'une solution de profit 7 qui n'est donc pas optimale,

l'approche gloutonne ne permet pas toujours de trouver la solution optimale.

Q12. Le tableau `T` à l'issue des lignes 2 à 8 :

```
[[0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0]]
```

après 1re itération : `T[1] = [0, 0, 3, 3, 3, 3]`
après 2ième itération : `T[2] = [0, 4, 4, 7, 7, 7]`
après 3ième itération : `T[3] = [0, 4, 4, 7, 7, 8]`.
La valeur renvoyée est 8, c'est le profit maximal pour ces 3 objets disponibles et la capacité 5.

Q13.
- les lignes 2 et 3 sont de complexité bornée ;
- les lignes 4 à 8 contiennent 2 boucles imbriquée de $n+1$ et $b+1$ tours et les tours sont de complexité bornée. Donc la complexité des lignes 4 à 8 est en $O(n \times b)$;
- de même, les lignes 9 à 14 sont de complexité en $O(n \times b)$.

Donc :

la complexité dans tous les cas est en $O(n \times b)$.

Q14. après la 1re itération : `S = [0, 0, 1], k = 2, r = 1`
après la 2ième itération : `S = [0, 1, 1], k = 1, r = 0` et la valeur `r = 0` provoque la sortie de boucle.
À chque tour de l'une des boucles la valeur de la variable `k` diminue de 1 et sa valeur initiale est $n-1$, donc en au plus $n-1$ tour la condition `k >= 0` provoque une sortie de boucle ; c'est à dire qu'il y a au plus $n-1$ tours de boucle et comme les tours sont de complexité bornée, les lignes 8 à 13 sont de complexité en $O(n)$.

la complexité n'est pas modifiée, elle reste en $O(n \times k)$.

Q15.

```
def estFeuille(a):
    return len(a['g']) == 0
```

Q16.

```
def possible(obj, S, b):
    m = len(S)
    n = len(obj)
    S0 = S + (n - m)*[0]
    S1 = S + (n - m)*[1]
    return contrainte(obj, S0, b) and profit(obj, S2) > Pmin
```

Remarque : si la fonction contrainte a été définie par une boucle sur la longueur de `S` comme proposé ici (et non de `obj`), on peut définir ainsi la fonction possible :

```
def possible(obj, S, b):
    m = len(S)
    n = len(obj)
    S1 = S + (n - m)*[1]
    return contrainte(obj, S, b) and profit(obj, S2) > Pmin
```

car `contrainte(obj, S, b)` calcule $\sum_{i=0}^{m-1} r_i x_i = \sum_{i=0}^{n-1} r_i x_i$ pour $r_m, \dots, r_{n-1} = 0$.

Q17.

```
elif possible(obj, arbre['S'], b):
    KPpse(arbre['g'], obj, b)
    KPpse(arbre['d'], obj, b)
```

Q18.

```
for i in range(len(T)):
    T[i] = rho * T[i]
```

Q19.

```
Pmax = 0
for k in range(len(S)):
    P = profit(obj, S[k])
    if P > Pmax:
        Pmax = P
        kmax = k
```

Q20.

```
DT = 1/(1 + Pbestofall - Pmax)
for i in range(n):
    if S[kmax][i] == 1:
        T[i] += DT
        T[i] = min(T[i], Tmax)
        # T[i] ne peut dépasser Tmax que si on lui ajoute DT
    T[i] = max(T[i], Tmin)
    # T[i] peut passer sous Tmin par évaporation
```

Q21.

```
15 : o0 = randint(0, n - 1)
17 : S[k][o0] = 1
18 : b2 = b2 - obj[o0][0]
```

Q22.

```
 3 : prob = {}
 5 : oi = candidats[i]
 6 : prob[oi] = T[oi]**\alpha*(b*obj[oi][1]/obj[oi][0])**beta
 9 : oi = candidats[i]
10 : prob[oi] = prob[oi] / s
```

Q23.

```
prob = construitProb(obj, candidats, b2, T)
x = choixCandidat(candidats, prob)
S[k][x] = 1
b2 = b2 - obj[x][0]
cadidats.remove(x)
miseAjourCandidats(obj, candidats, b2)
```

Q24. Les méthodes qui donnent la solution optimale sont : force brute, PSE et programmation dynamique. Parmi ces méthodes la programmation dynamique est la plus rapide et la force brute la plus lente.

L'algorithme ACO donne la solution optimale à partir de `nbit = 5` et donne le même profit que l'algorithme glouton pour `nbit = 1` (on ne profite pas des traces de phéromone, l'algorithme ACO n'est pas intéressant pour `nbit = 1`).

L'algorithme le plus rapide est l'algorithme glouton.

Pour l'algorithme ACO, on peut jouer sur le nombre de fourmis (5 dans les tests réalisés), plus le nombre de fourmis est élevé, meilleur sera le résultat, mais plus le temps d'exécution sera grand ; de même pour le nombre de nombre d'itération comme on le constate sur les tests réalisés. On peut également modifier les paramètres α, β et ρ, ce qui ne modifiera pas le temps d'exécution mais aura un impact sur le résultat obtenu.