Proposition de corrigé du sujet CCMP 2025, filière PSI.

version 2 du 4 décembre 2025.
Je n'ai pas testé tous les codes, en particulier ceux relatifs à l'algorithme ACO. Dès que j'aurai un peu de temps je posterai un fichier python avec les codes et des exemples pertinents de données.

Pour toute remarque et correction merci de m'écrire à MCanals@ac-nancy-metz.fr
(merci à François Chiaruttini pour la correction d'une erreur à la question 10)

1 Base de données - Exemple de fret maritime de conteneurs

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
WHERE idN>0
GROUP BY idN
```

Cette requête ne renvoie pas les identifiants des navires n'ayant aucun conteneur. Pour les obtenir dans la table résultat il faut effectuer une jointure à gauche (il me semble que ce n'est pas au programme) :

```
SELECT Navires.idN, count(idC)
FROM Navires LEFT JOIN Conteneurs ON Navires.idN = Conteneurs.idN
GROUP BY Navires.idN
```

2 Structure de données pour l'espace des objets

Q3.

```
def profit(obj, S) :
    p = 0
    for i in range(len(S)) :
        if S[i] == 1 :# on prend l'objet i
            p = p + obj[i][1]
    return p
```

Q4.

```
def contrainte(obj, S, b) :
    r = 0
    for i in range(len(S)) :
        if S[i] == 1 :# on prend l'objet i
            r = r + obj[i][0]
    return r <= b
```

3 Structure de données pour l'espace des solutions

Q5. Pour la feuille `b` la liste `S` est `[1,1,0]`
Pour la feuille `c` la liste `S` est `[1,0,1]`

Q6. Le nombre de feuilles est multiplié par deux chaque fois qu'on augmente n de 1. Lorsque $n = 0$ il y a une feuille. le nombre de feuilles est 2^n.
Un algorithme de type "force brute" calculerait le profit réalisé pour chaque feuille de l'arbre et vérifierait la contrainte. Le calcul du profit et la vérification de la contrainte nécessitent tous deux le parcours de la liste des objets. Pour chaque objet on effectue un test et au plus une addition. Ces deux fonctions ont donc une complexité en $O(n)$ et un algorithme "force brute" aurait une complexité en $O(n2^n)$.

4 Résolution approchée par un algorithme glouton

Q7. Dans le cas où `obj = [(2,3),(1,4),(4,4)]` et `b = 5` on a :

- Après la première étape : `Lqi = [1.5, 4.0, 1.0]` et `Li = [0, 1, 2]`
- Après la deuxième étape : `Lqi = [4.0, 1.5, 1.0]` et `Li = [1, 0, 2]`
- On sélectionne l'objet 1, de poids 1, b prend la valeur 4. On sélectionne l'objet 0 de poids 2, b prend la valeur 2. On ne peut pas sélectionner l'objet 2.

La liste est `S = [1, 1, 0]`

Q8.

```
1  def construitLi(obj):
2      Lqi = []
3      Li = []
4      for i in range(len(obj)):
5          Lqi.append(obj[i][1]/obj[i][0])
6          Li.append(i)
7      for i in range(1, len(Lqi)) :
8          x = Lqi[i]
9          j = i
10         while j>0 and Lqi[j-1] < x :
11             Lqi[j] = Lqi[j-1]
12             Li[j] = Li[j-1]
13             j -= 1
14         Lqi[j] = x
15         Li[j] = i
16     return Li
```

Q9. Le meilleur des cas est lorsque la liste est déjà triée par ordre décroissant, dans ce cas la boucle `while`, ligne 10, est vide et la complexité du tri est $O(n)$.
Le pire des cas est lorsque la liste est triée par ordre croissant, l'insertion de l'élément de rang i nécessite i comparaisons. la boucle `while` a pour complexité $O(i)$. Les autres lignes de la boucle `for` (8, 9, 14, 15) ont une complexité constante.
La complexité de la boucle `for` est donc $O\left(\sum_{i=0}^{n-1} i\right) = O(n^2)$. C'est la complexité de ce tri.

Q10.

```
S = len(obj)*[0]
Li = construitLi(obj)
j = 0
while b > 0 and j < len(obj) :
    if obj[Li[j]][0] <= b :   # obj[Li[j]] est le jème objet par ratio décroissant
        S[Li[j]] = 1              # On prend l'objet Li[j]
        b = b - obj[Li[j]][0]
    j += 1
```

Q11. Dans ce cas particulier, la solution optimale consiste à prendre les objets 1 et 2, c'est à dire `S = [0, 1, 1]`. On a alors un sac de valeur $4+4=8$.
La stratégie gloutonne ne calcule pas toujours une solution optimale.

5 Résolution exacte par un algorithme de programmation dynamique

Q12. À l'issue des lignes 2 à 8 on a :
`T = [[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0]]`
À la fin de chacune des trois itérations on a :
lorsque i = 0, `T[i+1] = [0, 0, 3, 3, 3, 3]`
lorsque i = 1, `T[i+1] = [0, 4, 4, 7, 7, 7]`
lorsque i = 2, `T[i+1] = [0, 4, 4, 7, 7, 8]`

Le programme renvoie la valeur 8, qui correspond à la valeur maximale d'un sac de capacité b=5, en utilisant le système d'objets `obj`.

Q13. Les lignes 2 et 3 ont une complexité constante.
La ligne 7 a une complexité constante donc la boucle interne a une complexité égale à $O(b+1) = O(b)$. Les lignes 5 et 8 ont une complexité constante, donc à chaque itération de la boucle externe, la complexité du code exécutée est égale à $O(b)$. La boucle comporte $n+1$ itération. Elle a donc une complexité en $O(nb)$. Finalement la complexité asymptotique de cette fonction est $O(nb)$.

Q14. Après la première boucle on a `S = [0,0,1]`, `k=1` et`r=5-4 = 1`.
Après la seconde boucle on a `S = [0,1,1]`, `k=0` et`r=1-1= 0`.

La complexité de ce bloc de code est $O(n)$ où n est le nombre d'objets puisque le code des lignes 11 à 13 est exécuté autant de fois qu'il y a d'objets dans le sac final, et la boucle `while`, ligne 9, comporte au maximum n itérations. La complexité de la fonction n'est pas modifiée.

6 Résolution exacte par un algorithme de programmation dynamique

Q15.

```
1  def estFeuille(a) :
2      return a['g'] == {}    # si le fils gauche est vide, le droit aussi.
```

Q16.

```
1  def possible(obj, Sk, b) :
2      n, k = len(obj), len(Sk)
3          return contrainte(obj,Sk + [0]*(n-k), b) and profit(obj,Sk + [1]*(n-k)) > Pmin
```

Q17.

```
1  def KPpse(arbre,obj,b) :
2      global Pmin, Sol
3      if estFeuille(arbre) :
4          if contrainte(obj, arbre['S'], b):
5              P = profit(obj, arbre['S'])
6              if P > Pmin :
7                  Pmin = P
8                  Sol = arbre['S']
9      else :
10         if possible(obj,arbre['g']['S'], b) :
11             KPpse(arbre['g'],obj,b)
12         if possible(obj,arbre['d']['S'], b) :
13             KPpse(arbre['d'],obj,b)
```

7 Résolution approchée par colonie de fourmis

Q18.

Q19.

Q20.

```
1  def mettreAjourTetSolution(obj, T, S) :
2      global SbestOfAll, PbestOfAll
3      # 2c)i
4      for i in range(len(T)) :
5              T[i] *= rho            # évaporation des traces de phéromone
6      # 2c)ii
7      kmax, Pmax = 0, profit(S[0])
8      for i in range(1, len(S)) :
9          p =  profit(S[i])
10         if p > Pmax :
11             kmax, Pmax = i, p
12     #2)c)iii
13     if Pmax>PbestOfAll :
14         PbestOfAll = Pmax
```

```
            SbestOfAll = S[kmax]
        #2)c)iv.v.vi.
        qtp = 1/(1 + PbestOfAll-Pmax)    # quantité de phéromone
        for i in range(len(obj)) :
            if SbesOfAll[i] == 1 :    # on prend l'objet i
                T[i] += qtp
        for i in range(len(T)) :
            if T[i] < Tmin :
                T[i] = Tmin
            elif T[i] > Tmax :
                T[i] = Tmax
```

Q21.

Q22.

Q23.

```

def construitProb(obj, candidats,b,T) :
    m = len(candidats)
    prob = {}
    for i in range(m) :
        oi = candidats[i]
        prob[oi] = T[oi]**alpha*(b*obj[oi][1]/obj[oi][0])**beta
    s = sum(prob.values())
    for i in range(m) :
        oi = candidats[i]
        prob[oi] = prob[oi] /s
    return s

n = len(obj)
T = [Tmax for i in range(n)]
PbestOfAll = 0
SbestOfAll = [0]*n
for it in range(nbit) :
    S = [[0]*n for i in range(nbAnts)]
    for k in range(nbAnts) :
        b2 = b
        o0 = randint(0,n-1)                                                    #2)b)i.
        S[k][o0] = 1                                                                #2)b)ii.
        b2 = b2 - obj[o0][0]
        candidats = [i for i in range(n) if i!= o0] #2)b)iii.
        miseAjourCandidats(obj, candidats,b2)
        while b2>0 and candidats != [] :                          #2)b)iv
            prob = construitProb(obj, candidats, b2, T)
            oi = choixCandidat(candidats, prob)
            S[oi] = 1
            candidats.remove(oi)
            b2 = b2 - obj[oi][0]
            miseAjourCandidats(obj, candidats,b2)
    mettreAjourTetSolution((obj, T, S)

```

8 Analyse des résultats

Q24. L'algorithme glouton est le plus rapide mais ne fournit pas une solution optimale. La solution fournie est assez proche de la solution optimale et le gain de temps est significatif (facteur 40) par rapport à la meilleure méthode donnant la solution optimal, à savoir la programmation dynamique.
Les méthodes force brute, Pse et programmation dynamique donne la solution optimale. Cette dernière s'avère bien plus efficace que l'algorithme Pse. A noter que dans le pire des cas l'algorithme Pse va envisager toutes les possibilités de l'arbre binaire.

L'algorithme ACO converge assez vite vers la solution optimale. Pour ce dernier on peut jouer sur le nombre d'itérations, le temps de calcul étant linéaire par rapport à ce paramètre. On peut aussi augmenter le nombre de fourmis, le temps de calcul est lui aussi linéaire par rapport à ce paramètre. On peut aussi jouer sur les paramètres `alpha` et `beta` pour donner plus ou moins de poids aux traces de phéromones.