TP (S1) 7 Algorithmes gloutons

Objectifs

- Découvrir la notion d'algorithme « glouton ».
- Étudier au travers de divers exemples, si un tel algorithme fournit une solution optimale à un problème proposé.

Fichier externe ?

NON *pas de fichier externe dans ce TP*

1 LE RENDU DE MONNAIE

Exercice 1 **Rendu de monnaie** [Sol 1] Le problème du rendu de monnaie est un problème d'algorithmique qui s'énonce de la façon suivante : étant donné un système de monnaie, comment rendre une somme donnée de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets ?

Dans la zone euro, le système en vigueur, en mettant de côté les centimes d'euros, met à disposition des pièces ou billets de : 1 €, 2 €, 5 €, 10 €, 20 €, 50 €, 100 €, 200 €, 500 €.

On suppose que nous avons à notre disposition un nombre illimité de ces pièces ou billets.

1. **[Premier contact]**

1.1) Supposons que la somme à rendre est de 7€. Faire la liste de toutes les façons de rendre une telle somme. Quel est le nombre minimal de pièces ou billets à rendre ?

1.2) Supposons que la somme à rendre soit maintenant de 463 €. Trouver une méthode permettant d'effectuer un rendu de monnaie en utilisant le moins possible de pièces sans avoir à énumérer toutes les possibilités. Combien de pièces ou de billets semblent nécessaires ?
Il est très probable qu'en résolvant cette dernière question, vous ayez mis en oeuvre la méthode dite « gloutonne » : tant qu'il reste quelque chose à rendre, on choisit la plus grosse pièce (ou plus gros billet) qu'on peut rendre (sans rendre trop).

2. **[Est-ce toujours une bonne stratégie ?]** Supposons que le système de monnaie mette à disposition uniquement des pièces de 1 €, 3 € et 4€ (on ne tient toujours pas compte des centimes d'euros) et supposons que nous devions rendre la somme de 6 €. Combien de pièces l'algorithme glouton nous amènerait-il à rendre ? Qu'en pensez-vous ? Pour le système monétaire européen (qui ne comporte aucune pièce ni aucun billet de 3€ ni de 4€), une telle situation ne se produirait pas ! Pour cette raison, un tel système de monnaie est qualifié de canonique.

Sans s'en rendre compte, tout individu met en oeuvre un algorithme glouton pour rendre la monnaie.

3. **[Mise en oeuvre d'une solution avec Python]** Un exemple incomplet de mise en oeuvre de l'algorithme avec python est proposé ci-dessous :

```
systeme_monnaie = [500,200,100,50,20,10,5,2,1]
# valeurs des pièces dans l'ordre décroissant
somme_a_rendre = 99    # somme à rendre
liste_pieces = []      # liste des pièces à rendre
i = 0                  # indice de la première pièce à rendre
```

```
while somme_a_rendre > 0:    # Construction de la liste des \
↪ pièces
    valeur = systeme_monnaie[i]
    if somme_a_rendre < valeur:
        i += 1
    else:
        ...
        # ajouter la pièce à la liste des pièces à rendre
        ...
        # Mettre à jour la nouvelle somme à rendre
return liste_pieces
```

3.1) Compléter les deux lignes en pointillé.

3.2) Écrire une fonction qui reçoit deux arguments (la somme à rendre et le système de monnaie) et qui renvoie la liste des pièces choisies par l'algorithme glouton. Testez cette fonction sur quelques exemples.

À retenir

L'algorithme glouton est un algorithme qui suit le principe de faire à chaque étape, un choix optimum local. L'algorithme agit en quelque sorte de manière « gloutonne » : devant un nouveau choix à faire, il choisit l'option qui apparaît être la meilleure à ce moment là. Cependant, comme cela a été vu dans la question 2, l'utilisation de cette méthode n'est pas toujours optimale.

2 LE PROBLÈME DU SAC À DOS

Exercice 2 **Le problème du sac à dos** [Sol 2] Le « **problème du sac à dos** » est un autre exemple classique de problème d'optimisation. Il peut s'énoncer ainsi : on dispose devant soi, d'objets ayant chacun une masse et une valeur, et d'un sac ne pouvant supporter plus d'une certaine masse. Il s'agit de remplir le sac en maximisant la valeur totale des objets et sans dépasser la masse maximale.

1. **[Premier contact]** Supposons que l'on dispose d'un sac à dos de contenance maximum 30 kg et de quatre objets A, B, C et D dont les caractéristiques sont les suivantes :

Objet	A	B	C	D
Masse	13 kg	12 kg	8 kg	10 kg
Valeur	70 €	40 €	30 €	30 €

1.1) Combien doit-on tester de combinaisons dans ce cas là pour remplir au mieux le sac à dos ?

1.2) Quelle semble être la solution optimale ?

2. **[Construction d'une solution réalisable]** Lorsque le nombre d'objets devient important, la solution précédente consistant à tester toutes les combinaisons devient inutilisable. En effet, s'il y a N objets, on peut démontrer qu'il y a $2^N - 1$ combinaisons à tester et, même pour un ordinateur très puissant, le temps de calcul devient trop important. On peut alors se tourner vers des méthodes qui fournissent rapidement une solution réalisable mais pas nécessairement optimale. C'est le cas de la méthode gloutonne. Pour cela, on introduit la notion « d'efficacité d'un objet » comme étant le rapport de sa valeur sur sa masse. Plus la valeur de l'objet est importante par rapport à sa masse, plus l'objet sera dit « efficace ».

2.1) Calculer l'efficacité de chacun des objets A, B, C et D evoqués précédemment, puis classer ces objets par ordre d'efficacité.

2.2) Proposer alors une méthode gloutonne permettant de résoudre le problème.

3. **[Préliminaire algorithmique]** On rappelle le code du tri rapide dans l'ordre croissant vu dans un précédent TP et qui s'applique à une liste de flottants.

```
def tri_rapide(L):
    """Renvoie le tri rapide de la liste L"""
    if len(L) < 2:
        return L
    else:
        Lg, Lm, Ld = [], L[0], []
        for x in L[1:]:
            if x < Lm:
                Lg.append(x)
            else:
                Ld.append(x)
        return tri_rapide(Lg) + [Lm] + tri_rapide(Ld)
```

Adapter le code précédent (en une fonction `tri_rapide_1d`) pour qu'il fonctionne pour des listes de quadruplets, et qu'elle trie selon la première coor-

donnée dans l'ordre décroissant. Par exemple, `tri_rapide_1d([[1.0, "a", 2, 3], [2.0, "b", 1, 1]])` devra renvoyer `[[2.0, "b", 1, 1], [1.0, "a", 2, 3]]`.

4. **[Mise en oeuvre d'une solution avec Python]** On suppose créée une variable `poids_max` qui contient la capacité maximale (en kilogrammes) du sac à dos. On suppose également construites 3 listes `liste_noms`, `liste_masses` et `liste_valeurs` qui contiennent respectivement les noms, masses et valeurs de chacun des objets. Par exemple, avec les objets utilisés précédemment :

```
# Liste des noms des objets
liste_noms = ["A","B","C","D"]
# Liste des masses des objets
liste_masses = [13,12,8,10]
# Liste des valeurs des objets
liste_valeurs = [70,40,30,30]
```

4.1) Créer une liste `L` dont le i-ème élément est une liste contenant respectivement l'efficacité du i-ème élément, son nom, sa masse et sa valeur.

4.2) Trier la liste `L` selon les efficacités par ordre décroissant.

4.3) Compléter alors les lignes en pointillé dans le script ci-dessous

```
liste_objet = [] # Initialisation de la liste des objets \
↪ rangés dans le sac
i = 0 # indice du premier objet
poids = 0 # somme des masses des objets déjà rangés
poids_max = 30
for i in range(len(L_triee)):
    if poids_max >=  ________________________ :
        liste_objet.append( _______________ )
        poids +=  _______________
```

Commenter le résultat obtenu par cette méthode gloutone.

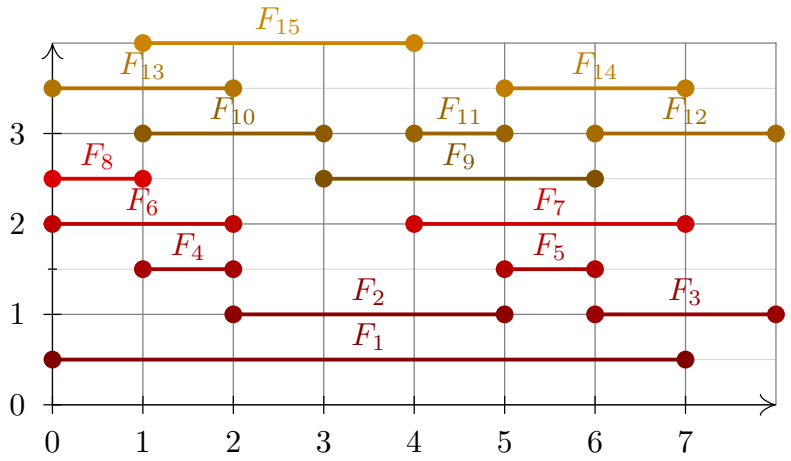
À retenir

A nouveau insistons sur le fait qu'en général, un algorithme glouton ne donne pas une solution optimale à un problème d'optimisation. Cependant, l'algorithme glouton fournit rapidement une solution réalisable.

3 LA FÊTE DU CINÉMA

Exercice 3 **La fête du cinéma (50mn)** [Sol 3] Le jour de la fête du cinéma, vous voulez aller voir un maximum de film dans la journée. Chaque film est caractérisé par un intervalle $[d, f[$ où d et f sont respectivement les heures de début et de fin. Deux films F_1 et F_2 sont alors compatibles si leur intervalle $[d1; f1[$ et $[d2; f2[$ sont disjoints. Quatre cinémas proposent des films :

- Le cinéma 1 propose 4 films :
  F1 : [3,4[F2 : [0,1[F3 : [2,3[F4 : [1,2[
- Le cinéma 2 propose également 4 films :
  F1 : [2,4[F2 : [0,1[F3 : [1,3[F4 : [0,2[
- Le cinéma 3 propose 3 films :
  F1 : [0,3[F2 : [1,2[F3 : [2,3[
- Le cinéma 4 propose une belle affiche composée de 15 films, représentée par le diagramme ci-dessous :



Nous allons envisager trois algorithmes gloutons différents, suivant nos priorités.

1. **[Un premier algorithme glouton]** Dans cette question, la priorité de l'algorithme glouton est d'aller voir le film qui commence en premier puis, une fois le film terminé, d'attendre le moins possible avant d'aller voir un autre film (dans le cas de films qui commencent à la même heure, c'est le film Fi avec le i minimal qui est choisi). Tester à la main cet algorithme glouton pour chacun des 4 cinémas.

2. **[Un deuxième algorithme glouton]** Dans cette question, la priorité de l'algorithme glouton est d'aller voir le film le plus court, puis une fois le film terminé, aller voir le film le plus court restant... (Là encore, dans le cas de films de même durée, c'est le film Fi avec le i minimal qui est choisi). Tester à la main cet algorithme glouton pour chacun des 4 cinémas.

3. **[Un troisième algorithme glouton]** Dans cette question, la priorité de l'algorithme glouton est d'aller voir le film qui se termine en premier.

3.1) Créer un cinéma aléatoire, c'est-à-dire, une fonction

```
Programmation(debut:int,fin:int,nb:int)->list
```

à qui on fournit une heure `debut` de début de séance, une heure `fin` de fin de séance et un nombre entier `nb` de films, et qui renvoie une liste aléatoire de `nb` éléments de la forme `["Fj",dj,fj]` où Fj désigne le nom du film, dj son horaire de départ et fj son horaire de fin, tous les films devant commencer et se terminer entre `debut` (inclus) et `fin` (inclus). *Indication : On pourra se servir de la commande `np.random.randint(a, b)` qui permet de choisir aléatoirement (selon une loi uniforme) un entier entre `a` inclus et `b` exclu.*

3.2) **[Préliminaire algorithmique]** Adapter à nouveau le code du tri rapide (en une fonction `tri_rapide_3c`) pour qu'il fonctionne pour des listes de triplets, et qu'elle trie selon la troisième coordonnée dans l'ordre croissant.

3.3) Écrire une fonction `Gloutons3(Prog:list)->list` à qui on fournit une liste (telle que décrite dans les questions précédentes) et qui renvoie une liste de films obtenus par la méthode gloutonne proposée. Tester cette fonction sur plusieurs exemples générés par la fonction `Programmation`

À retenir

En résumé :

- Un algorithme glouton est un algorithme utilisé pour résoudre un problème d'optimisation où l'on fait à chaque étape le « meilleur choix local » (charge au concepteur de l'algorithme de définir ce qu'est un meilleur choix local)
- Les algorithmes gloutons ne sont pas toujours adaptés : pour certains problèmes, ils ne permettent pas de donner une solution optimale.
- Lorsqu'un algorithme glouton permet de trouver une solution optimale à un problème, cet algorithme est souvent un excellent choix.

4 LE PROBLÈME DES STATIONS D'ESSENCE

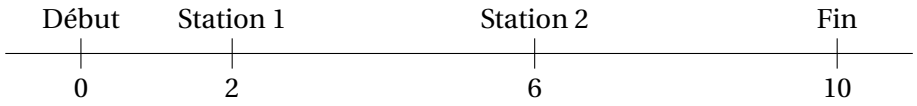
Exercice 4 Les stations d'essence [Sol 1]

Élaborer un algorithme glouton n'exclut pas d'utiliser en même temps d'autres techniques algorithmiques.

Dans cet exercice, nous allons proposer un exemple de mélange de stratégie gloutonne et de méthode « diviser pour régner ».

Un conducteur doit relier avec sa voiture deux villes distantes de N kilomètres, avec une seule route entre les deux.

Avec un plein, la voiture peut parcourir C kilomètres. Sur la route se trouvent des stations services, chacune étant caractérisée par un couple (x,p) où $x \in [\![0\,,\,N]\!]$ désigne la position de la station et p le prix du carburant (ramené au prix par kilomètre) proposé par cette station. On modélise cette situation par une liste de tuples. Par exemple, la situation suivante serait modélisée par la liste $[(0,0),(2,4),(6,3),(10,0)]$



Le prix du carburant serait donc de 4€ par kilomètre pour la station 1, et 3€ par kilomètre pour la station 2.

Le réservoir contient initialement une quantité Q (donnée en kilomètres) avec $Q \leq C$ de carburant.

L'objectif est de savoir à quelles stations il faut s'arrêter pour acheter son carburant afin de payer le moins possible pour faire le parcours ? (Il n'y a pas d'obligation de faire le plein à chaque arrêt).

On propose l'algorithme suivant, dont on pourrait montrer qu'il est optimal :

- S'il n'y a qu'une station service, y acheter juste l'essence qu'il faut pour terminer le parcours. Si la capacité du réservoir est insuffisante, l'algorithme renvoie une erreur (il n'y a pas de solution)
- S'il y a au moins deux stations services, on trouve la station (x_k,p_k) proposant l'essence la moins chère. On détermine alors récursivement la façon la moins chère d'aller de 0 à x_k en utilisant les stations situées entre 0 et x_k, puis à la station (x_k,p_k) :
  - soit on fait le plein si $N - x_k > C$ (on ne peut pas finir à partir de x_k) et on calcule récursivement la façon la moins chère d'aller de x_k à l'arrivée.
  - soit on remplit juste le carburant nécessaire pour finir le trajet si c'est possible.

1. Écrire une fonction `Minimum_Partiel(L:list,a:int,b:int)` à qui on fournit une liste de tuples (x_k,p_k) et deux entiers a et b, et qui renvoie k, un indice **strictement compris** entre a et b correspondant à la valeur minimale de p_k. Par exemple, l'appel à cette fonction avec `L = [(0,0),(50, 3), (100, 4), (150, 5), (200, 2), (250, 6), (300, 3), (350, 4), (400, 5), (450, 2),(500,0)]`, avec $a = 2$ et $b = 8$ devra renvoyer 4.

2. On propose ci-dessous le code de la fonction `itineraire` :

```
def itineraire(Q:int,C:int,deb:int,fin:int,st:list):
    """
    Paramètres d'entrée :
    Q : quantité de carburant disponible dans
        le réservoir, en kms
    C : capacité du réservoir
    deb : indice du tuple de départ dans stations
    fin : indice du tuple de d'arrivée dans stations
    st : liste de tuples stations service (x,p)
              (x abscisse, p prix du carburant) triée
              dans l'ordre croissant suivant x, complétée
              par le point de départ(0,0) et le point
              d'arrivée (N,0)

    Paramètre de sortie :
    un tuple formé de la liste des (indices,cout)
    correspondants aux stations choisies et le prix
    du plein qui y est fait, le carburant qu'il reste
    dans le réservoir à l'arrivée
    """
    if st[fin][0]-st[deb][0] <= Q:
        return [],Q-(st[fin][0]-st[deb][0])
    else:
        if len(st[deb+1:fin]) == 0 :
            print("Impossible de rejoindre la destination: \
            ↪ aucune solution")
            return None
        else:
            indice_etape = minimum_partiel(st,deb,fin)
            L1,q_dispo1 =itineraire(Q,C,deb,indice_etape,st)
            if st[fin][0]-st[indice_etape][0] <= C:
                    plein = \
                    ↪ st[fin][0]-st[indice_etape][0]-q_dispo1
```

```
31                      cout = plein*st[indice_etape][1]
32                      return (L1+[(indice_etape,cout)],0)
33              else:
34                  L2,q_dispo2 = \
                    ↪ itineraire(C,C,indice_etape,fin,st)
35                  cout = (C-q_dispo1)*st[indice_etape][1]
36                  return (L1+[(indice_etape,cout)]+L2,0)
```

2.1) La lecture des lignes 27 et 33 du programme permettent de dire que la fonction `itineraire` est de quel type ?

2.2) A quelle situation correspondent les lignes 19 et 20 du programme ?

2.3) A quelle situation correspondent les lignes 22 et 23 du programme ?

2.4) A quoi correspond la variable `q_dispo1` introduite à la ligne 27 ?

2.5) A quoi correspondent les deux situations envisagées dans les lignes 28 et 32 ?

2.6) Exécuter le programme avec les paramètres `Q=75`, `C=200`, `deb=0`, `fin=10` et `st=[(0,0),(50, 3), (100, 4), (150, 5), (200, 2), (250, 6), (300, 3), (350, 4), (400, 5), (450, 2),(500,0)]`.

2.7) Une fois récupéré le tuple de sortie, comment pourrait-on calculer le coût total du trajet ?

SOLUTIONS DES EXERCICES

Solution 1

1. **1.1)** On peut proposer les façons suivantes :
   1+1+1+1+1+1+1
   1+1+1+1+1+2
   1+1+1+2+2
   1+2+2+2
   1+1+5
   2+5
   Parmi toutes ces possibilités, la dernière est celle qui nécessite le moins de pièces ou de billets (seulement 2).

   1.2) On peut commencer par utiliser un billet de 200€. Il reste alors à rendre 263€.
   On peut alors utiliser un autre billet de 200€. Il reste alors 63€ à rendre.
   On choisit alors un billet de 50€. Il reste alors 13€ à rendre.
   On utilise alors un billet de 10€. Il reste alors 3€ à rendre.
   On utilise alors une pièce de 2€ puis une pièce de 1€.
   Au total, avec ce choix, 6 pièces ou billets ont été nécessaires.

2. L'algorithme décrit dans la question précédente nous conduirait à utiliser d'abord une pièce de 4€ puis 2 pièces de 1€. Ce n'est pas une solution optimale puisque cette solution demande d'utiliser 3 pièces alors qu'on peut rendre la monnaie avec seulement 2 pièces (3€ + 3€).

3. **3.1)** Les deux pointillés pourraient être respectivement remplacés par
   `liste_pieces.append(valeur)`
   et
   `somme_a_rendre=somme_a_rendre-valeur`.

   3.2) On peut proposer la solution suivante :

```
def rendu_monnaie(somme:int,systeme:list)->list:
    liste_pieces = [] #Liste des pièces à rendre
    somme_a_rendre=somme
    i=0  #indice de la première pièce à rendre
    while somme_a_rendre > 0: #Construction de la liste des \
    ↪ pièces
        valeur = systeme[i]
        if somme_a_rendre < valeur:
            i += 1
        else:
            liste_pieces.append(valeur)
            somme_a_rendre=somme_a_rendre-valeur
    return liste_pieces
```

Solution 2

1. **1.1)** On peut tester :

   - Les 4 objets en même temps : un seul test
   - 3 objets (A,B et C ; A,B et D ; A,C et D ; B, C et D) : 4 tests
   - 2 objets : 6 tests
   - 1 seul objet : 4 tests

   Cela fait donc au total 15 tests à effectuer.

   1.2) La solution optimale semble être de prendre les objets A et B : la somme de leur masse fait moins de 30 kg et la valeur totale de ces objets est de 110€.

2. **2.1)** Les efficacités des objets A, B, C et D sont respectivement égales à $\frac{70}{13} \approx 5,38$, $\frac{40}{12} \approx 3,33$, $\frac{30}{8} = 3,75$ et $\frac{30}{10} = 3$, ce qui donne :

   $$\text{efficacité(D)} < \text{efficacité(B)} < \text{efficacité(C)} < \text{efficacité(A)}$$

   2.2) On pourrait proposer la méthode gloutonne suivante : tant qu'il reste de la place dans le sac à dos, on choisit l'objet ayant la plus grande efficacité (et dont la masse ne fait pas dépasser la capacité maximale du sac).

3.
```
def tri_rapide_1d(L):
    """Renvoie le tri rapide de la liste L dans l'ordre \
    ↪ décroissant selon la 1ère coord"""
    if len(L) < 2:
        return L
    else:
        Lg, Lm, Ld = [], L[0], []
        for x in L[1:]:
            if x[0] < Lm[0]:
                Lg.append(x)
            else:
                Ld.append(x)
        return tri_rapide_1d(Ld) + [Lm] + tri_rapide_1d(Lg)
```

```
>>> tri_rapide_1d([[1.0, "a", 2, 3], [2.0, "b", 1, 1]])
[[2.0, 'b', 1, 1], [1.0, 'a', 2, 3]]
```

4. **4.1)** On peut proposer le script suivant :

```
liste_noms = ["A", "B", "C", "D"]
liste_masses = [13, 12, 8, 10]
liste_valeurs = [70, 40, 30, 30]
L = []
for i in range(len(liste_noms)):
    w = liste_valeurs[i]/liste_masses[i]
    x = liste_noms[i]
    y = liste_masses[i]
    z = liste_valeurs[i]
    L.append([w, x, y, z])
```

4.2) Il n'y a qu'à écrire :

```
>>> L_triee = tri_rapide_1d(L)
>>> L_triee
[[5.384615384615385, 'A', 13, 70], [3.75, 'C', 8, 30], \
↪ [3.3333333333333335, 'B', 12, 40], [3.0, 'D', 10, 30]]
```

4.3) On peut proposer le script suivant :

```
liste_objet = [] # Initialisation de la liste des objets \
↪ rangés dans le sac
i = 0 # indice du premier objet
poids = 0 # somme des masses des objets déjà rangés
poids_max = 30
for i in range(len(L_triee)):
    if poids_max >= poids + L_triee[i][2]:
        liste_objet.append(L_triee[i][1])
        poids += L_triee[i][2]
>>> poids
21
>>> liste_objet
['A', 'C']
```

La méthode gloutonne ne fournit pas dans notre exemple une solution optimale. En effet, nous avons vu plus haut que les objets A et B permettent d'atteindre une valeur de 110 € pour une masse de 25 kg, alors que la méthode gloutonne propose de prendre les objets A et C pour un montant total de 100 € et une masse totale de 21 kg.

Solution 3

1.
- Pour le cinéma 1, le film qui commence en premier est F2, puis une fois sorti, on peut aller voir F4, puis F3 puis F1.
- Pour le cinéma 2, F2 et F4 commencent en premier, donc, d'après la règle proposée, on va voir F2. Le film qui commence alors en premier est F3.
- Pour le cinéma 3, on va voir seulement F1.
- Pour le cinéma 4, on va voir seulement F1.

2.
- Pour le cinéma 1, les 4 films ont la même durée. D'après la règle proposée, on va donc voir F1 puis la séance est finie !
- Pour le cinéma 2, on va d'abord voir F2 puis F1.
- Pour le cinéma 3, on va d'abord voir F2 puis F3.
- Pour le cinéma 4, on commence par F4 puis F5 puis F3.

3. **3.1)** On peut proposer la fonction suivante (après avoir importé le module `numpy`) :

```
def Programmation(debut:int,fin:int,nb:int)->list:
    L = []
    for j in range(nb):
        dj = np.random.randint(debut,fin)
        fj = np.random.randint(dj+1,fin+1)
        Fj = "F"+str(j+1)
        L.append([Fj, dj, fj])
    return L
```

3.2) On peut utiliser le code ci-après.

```
def tri_rapide_3c(L):
    """Renvoie le tri rapide de la liste L dans l'ordre \
    ↪ croissant selon la 3ème coord"""
    if len(L) < 2:
        return L
    else:
        Lg, Lm, Ld = [], L[0], []
        for x in L[1:]:
            if x[2] < Lm[2]:
                Lg.append(x)
            else:
                Ld.append(x)
        return tri_rapide_3c(Lg) + [Lm] + tri_rapide_3c(Ld)
>>> tri_rapide_1d([["F1", 2, 3], ["F2", 4, 2]])
[['F2', 4, 2], ['F1', 2, 3]]
```

3.3) La fonction suivante convient :

```
def Gloutons3(Prog:list)->list:
    n = len(Prog)

    Prog_t = tri_rapide_3c(Prog)

    Planning = [Prog_t[0]] #Planning de la journée
    j = 0
    for i in range(1, n):
       if Prog_t[i][1] >= Prog_t[j][2]: #Si le film \
       ↪ commence après la fin du film j
           Planning.append(Prog[i])
           j = i # garde en mémoire l'indice (dans Prog_t) \
           ↪ du dernier film ajouté au planning
    return Planning
>>> Prog = Programmation(0, 10, 5)
>>> Prog
[['F1', 2, 6], ['F2', 3, 4], ['F3', 9, 10], ['F4', 9, 10], \
↪ ['F5', 0, 2]]
>>> tri_rapide_3c(Prog)
[['F5', 0, 2], ['F2', 3, 4], ['F1', 2, 6], ['F3', 9, 10], \
↪ ['F4', 9, 10]]
>>> Gloutons3(Prog)
[['F5', 0, 2], ['F2', 3, 4], ['F4', 9, 10]]
```

Solution 1

1.

```
def minimum_partiel(L:list,a:int,b:int):
    """
    Paramètres d'entrée :
    L : liste de tuples (x,p)
    a et b sont des entiers désignant deux indices de L

    Paramètre de sortie :
    l'indice de l'élément de valeur p minimale , indice \
    ↪ strictement compris entre a et b
    """
    indice = a+1
    minimum = L[a+1][1]
    for k in range(a+2,b):
        if L[k][1] < minimum:
            minimum = L[k][1]
            indice = k
    return indice
```

2. **2.1)** Il s'agit d'une fonction récursive.

2.2) Les lignes 19 et 20 correspondent au cas où la distance à parcourir est inférieure à la capacité du réservoir, et où le trajet pourra être effectué sans s'arrêter à une station intermédiaire.

2.3) Les lignes 22 et 23 correspondent au cas où la distance à parcourir est strictement supérieure à la capacité du réservoir mais qu'il n'y a pas de station intermédiaire où un plein puisse être effectué.

2.4) La variable `q_dispo1` correspond au carburant qu'il reste dans le réservoir lorsqu'on arrive à la station intermédiaire d'indice `indice_etape`.

2.5) La ligne 28 correspond à la situation où le plein à la station d'indice `indice_etape` suffira à terminer le trajet. La ligne 32 correspond à la situation où il faudra encore faire un plein intermédiaire entre la station d'indice `indice_etape` et la fin du trajet.

2.6) Le programme renvoie le tuple `[(1,375),(4,400),(6,150),(9,100)],0)`.

2.7) Le coût total pourrait être obtenu en sommant les secondes composantes de la liste (elle-même étant la première composante du tuple), c'est-à-dire :

```
T = itineraire(Q,C,deb,fin,stations)
Cout = 0
Liste = T[0]
for k in range(len(Liste)):
    Cout = Cout+Liste[k][1]
```