éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

LE PROBLÈME DU SAC À DOS

- Histoire de l'informatique
- Représentation des données
- Traitement des données
- Interactions entre l'homme et la machine sur le Web
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

1. Introduction

Diverses activités peuvent être menées avec les élèves sur le problème du sac à dos. C'est en effet l'occasion de réinvestir plusieurs notions étudiées au cours de l'année de première. Ces activités peuvent aussi se placer au cœur d'un projet qui sera conduit durant l'année en classe de première. De plus la prolongation de ce travail peut être envisagée en classe de terminale en mobilisant de nouvelles notions permettant d'autres méthodes de résolutions.

1.1. Le problème

Le problème type

On dispose d'un sac pouvant supporter un poids maximal donné et de divers objets ayant chacun une valeur et un poids. Il s'agit de choisir les objets à emporter dans le sac afin d'obtenir la valeur totale la plus grande tout en respectant la contrainte du poids maximal. C'est un *problème d'optimisation avec contrainte*.

Ce problème peut se résoudre par *force brute*, c'est-à-dire en testant tous les cas possibles. Mais ce type de résolution présente un problème d'efficacité. Son coût en fonction du nombre d'objets disponibles croît de manière exponentielle.

Nous pouvons envisager une *stratégie gloutonne*. Le principe d'un algorithme glouton est de faire le meilleur choix pour prendre le premier objet, puis le meilleur choix pour prendre le deuxième, et ainsi de suite. Que faut-il entendre par meilleur choix ? Est-ce prendre l'objet qui a la plus grande valeur, l'objet qui a le plus petit poids, l'objet qui a le rapport valeur/poids le plus grand ? Cela reste à définir.

Le problème à résoudre

Nous disposons d'une clé USB qui est déjà bien remplie et sur laquelle il ne reste que 5 Go de libre. Nous souhaitons copier sur cette clé des fichiers vidéos pour l'emporter en voyage. Chaque fichier a une taille et chaque vidéo a une durée. La durée n'est pas proportionnelle à la taille car les fichiers sont de format différents, certaines vidéos sont de grande qualité, d'autres sont très compressées.
Le tableau qui suit présente les fichiers disponibles avec les durées données en minutes.

Fichier	Durée	Taille
Video 1	114	4,57 Go
Video 2	32	630 Mo
Video 3	20	1,65 Go
Video 4	4	85 Mo
Video 5	18	2,15 Go
Video 6	80	2,71 Go
Video 7	5	320 Mo

Le nombre de fichiers est volontairement réduit pour l'exemple. On peut ensuite augmenter ce nombre ainsi que la place disponible sur la clé.

1.2. Les notions

- Représentation de l'information (video, format, compression).
- Stockage en mémoire (différence entre la taille d'un fichier et la taille occupée en mémoire).
- L'octet comme unité de mesure de la taille d'un fichier (Ko, Mo, Go).
- Les types int, float, str.
- Ecriture binaire d'un entier, conversion avec l'utilisation de chaînes de caractères.
- Utilisation de types construits pour représenter les données : listes, p-uplets, dictionnaires.
- Algorithmes de parcours de listes ou de dictionnaires.
- Programmation d'un algorithme de tri (tri par insertion ou tri par sélection) ou utilisation d'une fonction du langage Python.
- Programmation d'un algorithme glouton.

Un prolongement en terminale pourra inclure les notions de graphe, d'arbre binaire et de programmation dynamique.

2. Représentation des données

Les tailles sont converties en Go. La taille d'un fichier et sa taille en mémoire ne sont pas distinguées.

2.1. Utilisation d'une liste de p-uplets

Un fichier est représenté par un triplet contenant son nom de type str, sa durée de type int et sa taille de type float. Les triplets obtenus sont les éléments d'une liste.

```
videos = [('Video 1', 114, 4.57), ('Video 2', 32, 0.630),
          ('Video 3', 20, 1.65), ('Video 4', 4, 0.085),
          ('Video 5', 18, 2.15), ('Video 6', 80, 2.71),
          ('Video 7', 5, 0.320)]
```

2.2. Utilisation d'un dictionnaire

L'ensemble des fichiers est représenté par un dictionnaire dont les clé sont les noms des fichiers de type str. Pour chaque clé, (chaque fichier), la valeur correspondante est un dictionnaire dont les clés sont les chaînes `"Durée"`et `"Taille"` de type str. Les valeurs correspondantes sont la durée de type int et la taille de type float.

```
videos = {'Video 1': {'Durée': 114,
                      'Taille': 4.57},
          'Video 2': {'Durée': 32,
                      'Taille': 0.630},
          'Video 3': {'Durée': 20,
                      'Taille': 1.65},
          'Video 4': {'Durée': 4,
                      'Taille': 0.085},
```

```
          'Video 5': {'Durée': 18,
                      'Taille': 2.15},
          'Video 6': {'Durée': 80,
                      'Taille': 2.71},
          'Video 7': {'Durée': 5,
                      'Taille': 0.320}
         }
```

Pour la suite nous utilisons la représentation sous la forme d'une liste. Il est bien sûr très formateur de demander aux élèves de faire la transposition des programmes qui suivent dans le cadre de l'utilisation d'un dictionnaire. Une difficulté importante est que les éléments d'un dictionnaire ne sont pas ordonnés. Il peut être judicieux de commencer par construire la liste des clés.

3. Force brute

Le principe est simple, il faut tester tous les cas possibles. La mise en œuvre l'est moins. Comment obtenir tous les cas sans les répéter et sans en oublier un ? Cette question pose la difficulté principale.
Une méthode est d'associer le chiffre 1 à un fichier s'il est choisi et le chiffre 0 sinon. Nous obtenons ainsi un nombre entier écrit en binaire avec 7 chiffres. Le nombre 1001100 signifie que nous avons choisi les fichiers 1, 4 et 5. Le nombre 1111111 signifie que nous avons choisi tous les fichiers. A chaque nombre correspond exactement une possibilité pour construire une partie de l'ensemble des 7 fichiers. Il apparaît alors que le nombre total de cas est 2^7 puisqu'avec 7 chiffres nous pouvons écrire exactement 2^7 nombres.

3.1. Ensemble des parties

Il s'agit donc d'écrire une fonction qui prend en paramètres un ensemble, (dans notre problème de 7 fichiers), et renvoie l'ensemble des parties qui peuvent être constituées.
Nous avons besoin de l'écriture d'un nombre en binaire. Soit nous programmons une fonction, soit nous utilisons la fonction **bin** de Python.

Remarque : la fonction **bin** renvoie une chaîne de caractères. Par exemple, **bin**(13) a pour valeur "0b1101". Nous devons donc retirer les deux premiers caractères "0b" de cette chaîne. L'instruction `bi = bin(13)[2:]` effectue cette tâche mais le slicing est à la limite du programme. Une autre possibilité est d'écrire une boucle pour constituer une nouvelle chaîne en ne prenant que les caractères à partir de l'indice 2. Mais il est aussi rapide d'écrire notre propre fonction de conversion.

```
def int_to_bin(n, nb):
    """ n et nb sont de type int
        n est le nombre à convertir en binaire
        nb est le nombre de bits utilisés """
    ch = ""
    while n > 0:
        r = n % 2
        n = n // 2
        ch = str(r) + ch
    ch = (nb - len(ch)) * "0" + ch
    return ch
```

Nous pouvons alors écrire une fonction `ens_des_parties` qui prend en paramètre un ensemble d'objets et renvoie une liste dont chaque élément est une partie de l'ensemble.

```
def ens_des_parties(ensemble):
    """ ensemble est une liste de p-uplets """
    nb = len(ensemble) # nombre d'éléments
    n = 2 ** nb # nombre de parties
    parties = [] # l'ensemble des parties
```

```
    for i in range(1, n):
        ch = int_to_bin(i, nb) # écriture de i sur nb bits
        partie = [] # construction d'une partie
        for j in range(len(ch)):
            if ch[j] == "1":
                partie.append(ensemble[j])
        parties.append(partie) # la partie construite est ajoutée à la liste
    return parties
```

3.2. Fonctions annexes

Pour chaque partie, nous devons calculer la durée totale et la taille totale des fichiers constituant la partie.

```
def duree_totale(liste):
    d = 0
    for triplet in liste:
        d += triplet[1]
    return d
```

```
def taille_totale(liste):
    t = 0
    for triplet in liste:
        t += triplet[2]
    return t
```

3.3. Programme final

Pour le programme final, nous écrivons une fonction `recherche` qui prend en paramètres un ensemble de parties et la contrainte, et renvoie la partie correspondant au meilleur choix satisfaisant la contrainte après avoir parcouru toutes les parties.

```
def recherche(ens_parties, contrainte):
    duree_max = 0
    solution = []
    for partie in ens_parties: # un choix possible de fichiers
        duree = duree_totale(partie)
        taille = taille_totale(partie)
        if taille <= contrainte and duree > duree_max:
            duree_max = duree
            solution = partie
    return solution, duree_max
```

La fonction `force\_brute` :

```
def force_brute(fichiers, taille_max):
    parties = ens_des_parties(fichiers)
    return recherche(parties, taille_max)

choix = force_brute(videos, 5)
duree_totale = choix[1]
choix_fichiers = [fichier[0] for fichier in choix[0]]
print(choix_fichiers, duree_totale)
```

La réponse est l'ensemble ['Video 2', 'Video 3', 'Video 6'], avec la valeur totale 132.

4. Algorithme glouton

4.1. Quelques fonctions

Un fichier vidéo est représenté par une liste comme `['Video 1', 114, 4.57]`.

Nous avons besoin de trois fonctions prenant en paramètre un tel fichier et renvoyant soit la durée, soit l'inverse de la taille (si la taille est minimale, son inverse est maximale), soit le rapport durée/taille.

```
def duree(fichier):
    return fichier[1]

def taille(fichier):
    return 1 / fichier[2]

def rapport(fichier):
    return fichier[1] / fichier[2]
```

4.2. Programme glouton

Fonction sorted

Nous définissons une fonction `glouton` qui prend en paramètres une liste de fichiers, une taille maximale (celle que peut stocker la clé USB) et le type de choix utilisé (par durée, par taille, ou par durée/taille).

Nous commençons par construire une nouvelle liste en triant la liste passée en paramètre suivant le type de choix utilisé avec la fonction `sorted`. L'instruction `help(sorted)` écrite dans l'interpréteur permet d'obtenir des informations sur la fonction `sorted`. Nous utilisons l'ordre décroissant.

La liste triée est parcourue et les noms des fichiers sont ajoutés un par un dans la variable `reponse`, tant que la taille totale ne dépasse pas la taille maximale. La durée totale et la taille totale sont stockés dans deux variables nommées `totale\_duree` et `taille`.

```
def glouton(liste, taille_max, choix):
    triee = sorted(liste, key=choix, reverse=True)
    reponse = []
    totale_duree = 0
    taille = 0
    i = 0
    while i < len(liste) and taille <= taille_max:
        nom, d, t = triee[i] # nom, durée et taille
        if taille + t <= taille_max:
            reponse.append(nom)
            taille += t
            totale_duree += d
        i += 1
    return reponse, totale_duree
```

Nous exécutons la fonction `glouton` en précisant le type de choix utilisé.

```
print(glouton(videos, 5, duree))
print(glouton(videos, 5, taille))
print(glouton(videos, 5, rapport))
```

Nous obtenons les résultats suivant le type de choix :

- soit par durée : `(['Video 1', 'Video 4', 'Video 7'], 123)`;
- spot par taille : `(['Video 7', 'Video 4', 'Video 2', 'Video 3', 'Video 5'], 79)`,
- soit par durée/taille : `(['Video 2', 'Video 7', 'Video 6', 'Video 4'], 121)`.

Nous constatons que le critère de durée est le plus intéressant, puisque la valeur totale est 123. Cette solution n'est cependant pas la solution optimale trouvée par force brute `['Video 2', 'Video 3', 'Video 6']` 132.

Tri par insertion

Un tri par insertion avec ordre décroissant (la liste initiale est modifiée en place) :

```
def tri_insertion(liste, choix):
    for i in range(len(liste)-1):
        k = i+1
        cle = liste[k]
        while k > 0 and choix(cle) > choix(liste[k-1]):
            liste[k] = liste[k -1]
            k = k - 1
        liste[k] = cle
```

Le programme glouton avec lequel nous obtenons bien sûr les mêmes résultats que précédemment :

```
def glouton(liste, taille_max, choix):
    tri_insertion(liste, choix) # modification ici
    reponse = []
    totale_duree = 0
    taille = 0
    i = 0
    while i < len(liste) and taille <= taille_max:
        nom, d, t = liste[i] # modification ici
        if taille + t <= taille_max:
            reponse.append(nom)
            taille += t
            totale_duree += d
        i += 1
    return reponse, totale_duree
```