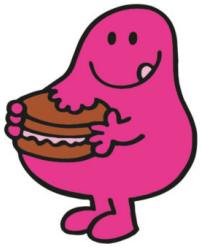
TD

TD n°16 : Les algorithmes gloutons - Révision 1ère	Thème 4 : Algorithmique
COURS et EXERCICES	

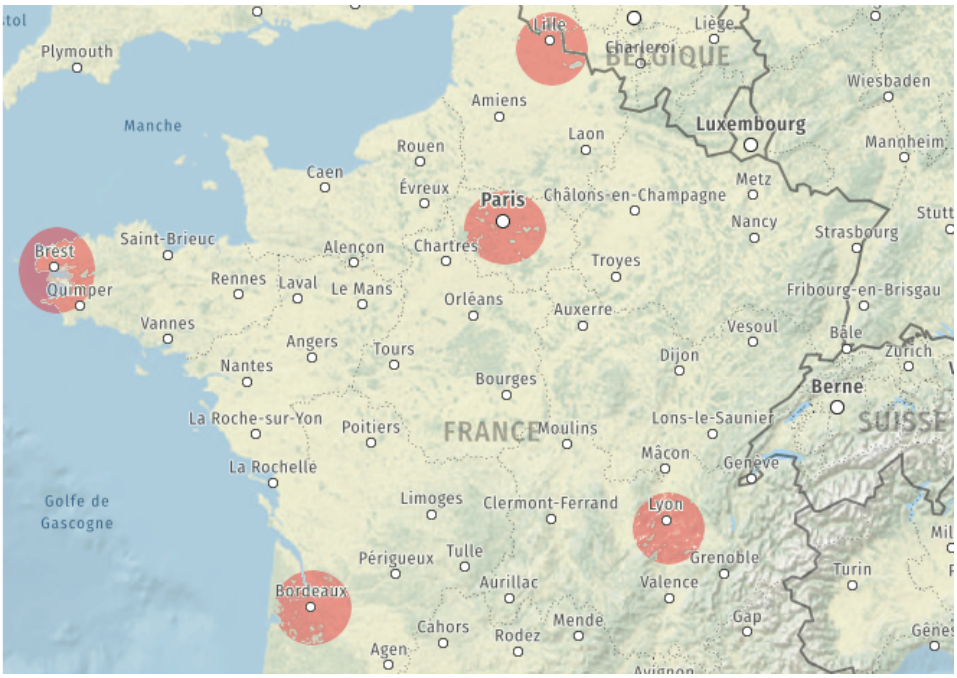


I. Problème d'optimisation : Le voyageur

Enoncé du problème

> Imaginons le problème suivant : vous devez organiser la tournée de votre commercial. Il doit passer dans toutes les villes sous sa responsabilité (une fois uniquement) et revenir à son point de départ (Lille pour notre exemple). Pour limiter les frais, il faut définir le trajet le moins long au total.

Nous allons nous limiter à 5 villes uniquement ici.



Voici le tableau des distances :

	Brest	Bordeaux	Lille	Lyon	Paris
Brest	-	598	708	872	572
Bordeaux	598	-	802	520	554
Lille	708	802	-	650	225
Lyon	872	520	650	-	465
Paris	572	554	225	465	-

Question 1 :
Partons de Lille. Combien de destinations différentes peut-on choisir ? Combien de "chemin" possible ?

Le problème se ramène à trouver un ordre de visite des quatre villes pour lequel la somme des distance données par ce tableau est aussi petite que possible.
Une manière simple d'aborder le problème consiste à énumérer tous les cas possibles et calculer la distance correspondante pour chacun des cas.

Réponse :

Question 2 : Résolution par force brute

Traiter le problème en faisant tous les cas de figure, à résumer dans un tableau et donner l'itinéraire le plus court

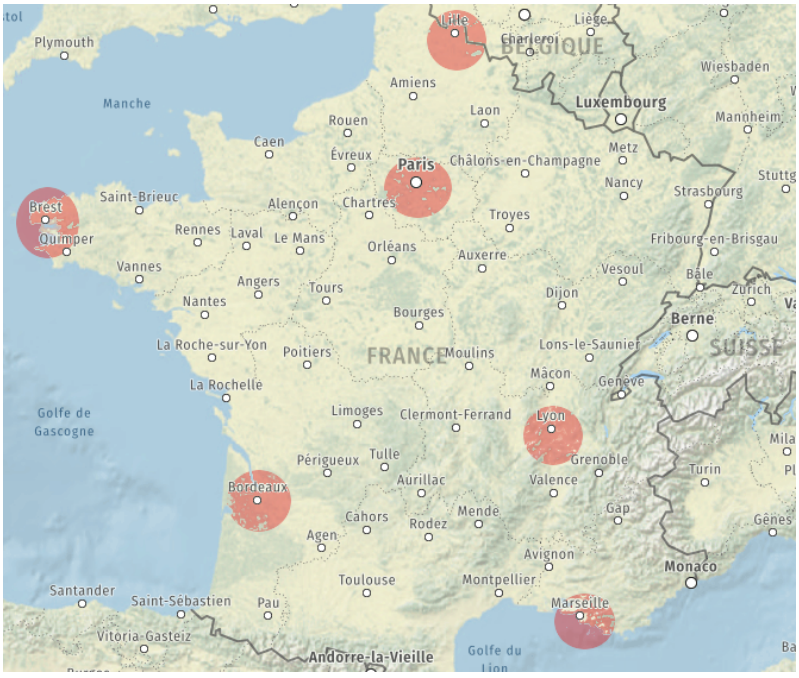
Afficher tous les chemin avec la distance totale 1. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 2. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 3. Lille -> ... -> ...-> ...-> ...-> ...-> Lille avec une distance totale de ... km 4. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 5. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 6. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 7. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 8. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 9. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 10. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 11. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km 12. Lille -> ... -> ...-> ...-> ...-> Lille avec une distance totale de ... km

Remarque
Les douze itinéraires correspondent chacun à l'un des douze itinéraire emprunté dans le sens inverse.

Question 3 :
Répondre au problème : quel est le trajet optimal. ?

Réponse

Question 4 :
Reprenons le même problème en rajoutant Marseille.

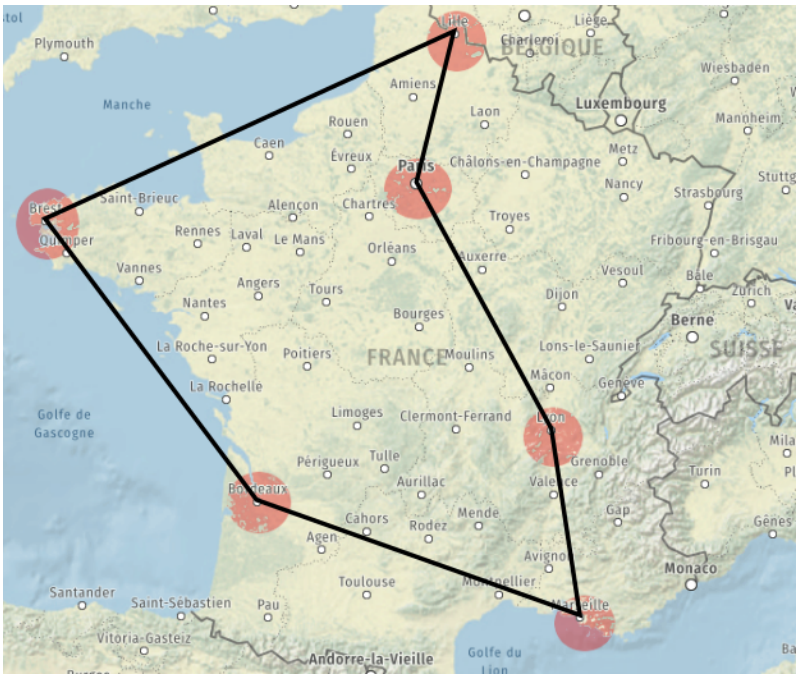


Question 5 :
Dans ce cas, avec 6 villes, combien de cas faut-il étudier ? Qu'en pensez-vous ?

Réponse :

Réponse au problème :

> Le plus petit trajet sera réalisé avec le parcours suivant (dans un sens ou dans l'autre) :
> Lille -> Paris -> Lyon -> Marseille -> Bordeaux -> Brest -> Lille en 3 000 km



Cette technique (répertorier tous les cas de figure et faire une étude exhaustive) n'est pas possible à grande échelle.
Déterminer le nombre d'itinéraires possibles : * 4 villes (hors ville de départ) on a : $\frac{4 \times 3 \times 2 \times 1}{2} = 12$ itinéraires possibles. * 10 villes (hors ville de départ), nombre d'itinéraires à tester : $\frac{10 \times 9 \times 8 \times 7 \cdots \times 2 \times, 1}{2} =$ 1 714 400 * 13 villes (hors ville de départ), nombre d'itinéraires à tester : $\frac{13 \times 12 \cdots \times 2 \times, 1}{2} =$ 3 113 510 400 * 20 villes (hors ville de départ), nombre de parcours à tester : $\frac{20 \times 19 \times 18 \times 17 \cdots \times 2 \times, 1}{2} =$ 1 216 451 004 088 320 000

> Face à de tels problèmes d'optimisation impossible à explorer exhaustivement, il peut être utile de connaître des algorithmes donnant rapidement une réponse qui, sans être nécessairement optimale, resterait bonne.

La méthode gloutonne que l'on va abordée dans ce TP donne une approche simple pour concevoir de tels algorithmes souvent approximatifs mais rapides.

II. Algorithmes Gloutons

Nous avons vu que la force brute ne permet pas de résoudre (en un temps raisonnable) le problème du voyageur de commerce lorsqu'on augmente le nombre de villes.

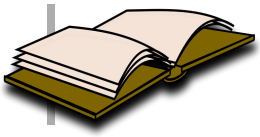
Maintenant que vous avez manipulé ce problème et vu à quel point il est complexe à résoudre, nous allons voir comment faire autrement.

Remarque : **greedy algorithms** en anglais, l'adjectif "greedy" signifiant avare/glouton.

Les **algorithmes gloutons** sont utilisés pour répondre à des ***problèmes d'optimisation***, c'est-à-dire des problèmes algorithmiques dans lesquels l'objectif est de trouver une solution " la meilleure possible " selon un critère, parmi un ensemble de solutions également valides mais potentiellement

moins avantageuses.
Le contexte général d'un tel problème d'optimisation est donc le suivant : * on considère un problème possédant un très grand nombre de solutions * on dispose d'une fonction mathématique évaluant la qualité de chaque solution * on cherche une solution qui soit bonne, voire meilleure.

Les algorithmes gloutons s'appliquent lorsque de plus : * la recherche d'une solution peut se ramener à une succession de choix qui produisent et précisent petit à petit une solution partielle * on dispose d'une fonction mathématique évaluant la qualité de chaque solution partielle (dont on attend qu'elle soit cohérente avec la fonction d'évaluation des solutions complètes).



Le **principe d'un algorithme glouton** est le suivant : * résoudre un problème étape par étape * à chaque étape, faire le choix optimal de moindre coût (de meilleur gain)

Le choix effectué à chaque étape n'est jamais remis en cause, ce qui fait que cette stratégie permet d'aboutir rapidement à une solution au problème de départ. C'est en ce sens que l'adjectif greedy (glouton/avare) caractérise ces algorithmes : ils terminent rapidement (glouton) sans fournir beaucoup d'efforts (avare).

III. Résolution approchée du problème du voyageur

III.1. Appliquons la méthode gloutonne à notre problème de voyageur

Le problème peut se résumer à « quelle sera ma prochaine étape ».

⚙**Méthode gloutonne :**
partant de la ville de départ, aller à la ville la plus proche, puis la ville la plus proche non visitée etc...

Question 6 :
Mettez en oeuvre cette méthode en partant de Lille.

Réponse :
Notre voyage fera donc **.....** kilomètres.
L'itinéraire ainsi obtenu est plus long que le circuit minimal de **.....** kilomètres vu en introduction, mais reste loin des assez nombreuses mauvaises solution qui demandaient plus de mille kilomètres. Et surtout, nous n'avons analyse qu'un unique itinéraire !

Mise en oeuvre : Python

Principe du Programme : + Les villes sont individuellement sous forme de dictionnaires avec les autres villes en clé et les dis- tances correspondantes en valeurs. Un autre dictionnaire est créé pour regrouper l'ensemble des villes :

```
''' Nous créons un dictionnaire les_villes qui va contenir des dictionnaires pour stock
distances entre villes '''

brest = {'Bordeaux':598, 'Lille':708, 'Lyon':872, 'Marseille':1130, 'Paris':572}
```

```
bordeaux = {'Brest':598, 'Lille':802, 'Lyon':520, 'Marseille':637, 'Paris':554}
lille = {'Brest':708, 'Bordeaux':802, 'Lyon':650, 'Marseille':1002, 'Paris':225}
lyon = {'Brest':872, 'Bordeaux':520, 'Lille':650, 'Marseille':367, 'Paris':465}
marseille = {'Brest':1130, 'Bordeaux':637, 'Lille':1002, 'Lyon':367, 'Paris':777}
paris = {'Brest':572, 'Bordeaux':554, 'Lille':225, 'Lyon':465, 'Marseille':777}
les_villes = {'Brest':brest, 'Bordeaux':bordeaux, 'Lille':lille, 'Lyon':lyon, 'Marseill
```

- Une fonction `plus_proche_voisin(ville,les_villes_voisines,Non_Visitees)` , qui renvoie le nom de la ville non encore visitée la plus proche ainsi que la distance correspondante.

```
def plus_proche_voisin(ville,les_villes_voisines,Non_Visitees):
       pass

ville='Lille'
print(plus_proche_voisin(ville,les_villes[ville],les_villes.keys()))
```

- Une fonction principal `trajet` qui prend en paramètre la ville de départ,le dictionnaire des villes à visiter.

```
def trajet(depart,avisitees):
    Nvisitees=....  #on crée un tableau avec les villes à visiter
    taille=len(avisitees)
    ville=depart
    parcours=[depart]  # le parcours débute par la ville choisie
    distance=0         # au départ la distance aprcourue est 0
    for i in range(.....):
        ...........   #on enleve la ville déja visitée de la liste

        ville,dist=plus_proche_voisin(ville,avisitees[ville],Nvisitees) #on appelle la

        distance....    #la distance parcourue augment de la distance entre les deux vi

        parcours.append(ville)  # On ajoute la ville ainsi visitée dans le tableau parc

        print(f"ville suivante {ville} avec une distance de {dist} km soit une distance

    ..... #la boucle est finie, on rajoute la ville de départ qui devient la ville de f

    distance+=les_villes[ville][depart]  #on ajoute la dernière distance à parcourir po

    print(f"ville suivante {ville} avec une distance de {dist} km soit une distance tot

    return parcours,distance

print(trajet('Lyon',les_villes))
```

Tester ce code avec d'autres villes de départ.

Qualité de l'approximation

- La solution donnée par l'algorithme glouton n'est pas nécessairement optimale. Peut-on néanmoins s'attendre à un certain niveau de qualité ?
- Si oui, ce niveau de qualité attendu est-il garanti, c'est-à-dire respecte même dans le pire des scénarios ?
- Ou bien est-il seulement hautement probable, c'est-à-dire généralement respecté mais avec des exceptions ?

Les réponses à ces questions dépendent fortement du problème considéré.
Pour notre problème du voyageur, ii a été démontré que lorsque le nombre de villes devient grand, le rapport entre la. solution gloutonne et la solution optimale est dans le pire des cas proportionnel **au logarithme du nombre de villes**.

Voir une simulation

IV Problème : Rendu de monnaie

Nous allons étudier un problème d'optimisation classique : le problème du rendu de monnaie de manière optimale. On cherche à rendre la monnaie avec un nombre minimal de pièces et billets.

Voici notre système de monnaie (exprimé en euros) : * Pièces : 0,01 ; 0,02 ; 0,05 ; 0,1 ; 0,2 ; 1 ; 2 * Billets : 5 ; 10 ; 20 ; 50 ; 100 ; 200; 500

On cherche par exemple à rendre 53 euros. On peut dans un tableau énumérer quelques solutions possibles et choisir celle qui minimise le nombre de pièces et de billets.

Rendus de monnaie	Nombre de pièces et de billets
$5\ 300 \times 0.01$	5 300
53×1	53
$5 \times 10 + 1 \times 2 + 1 \times 1$	7
$2 \times 20 + 3 \times 5 + 1 \times 2$	6
$1 \times 50 + 1 \times 2 + 1 \times 1$	3

L'utilisation de **ce type de méthode** est **très coûteux en temps de calcul** car il faut explorer toutes les possibilités.

IV.1. L'algorithme glouton

On procède étape par étape en faisant, à chaque étape, le meilleur choix possible. On ne remet jamais en cause les choix faits aux étapes passées. Dans notre cas nous allons rendre la monnaie en commençant par la pièce ou le billet avec la plus grande valeur possible (en restant inférieur à la somme à rendre). Cela correspond à notre dernière ligne de tableau ($1 \times 50 + 1 \times 1 + 1 \times 2$€).
On recommence ainsi jusqu'à obtenir une valeur nulle.
On note : * systeme : liste des pièces et des billets * somme : montant à obtenir * somme_restante : montant qui reste à rendre * monnaie : liste qui contient les valeurs rendues

```
1. initialiser monnaie à une liste vide
2. initialiser la somme_restante à somme
3. tant que somme_restante >0 :
    * On choisit la plus grande valeur dans systeme inférieure à somme_restante
```

```
    * on ajoute cette valeur à monnaie
    * on ajoute cette valeur à somme_restante
4. renvoyer monnaie
```

Reprenons notre exemple avec somme= 53 et systeme= $[0.01, 0.02, 0.05, 0.1, 0.2, 1, 2, 5, 10, 20, 50]$.

Voici la liste des étapes :

Initialisation	monnaie=[]	somme_restante=53
étape 1	monnaie=$[50]$	somme_restante=3
étape 2	monnaie=$[50, 2]$	somme_restante=1
étape 3	monnaie=$[50, 2, 1]$	somme_restante=0

Notre solution dépend du nombre de pièces et de billets disponibles. Si nous sommes limités sur certaines pièces et/ou certains billets, les résultats seront différents.

Exercice 1 :

Traiter l'exercice précédent avec pour système code S = $[1, 2, 5, 50, 100]$ et pour somme : 27 € .

Réponse :

IV.2. Solution et solution optimale

Pourquoi notre système monétaire ne possède pas de pièces ou de billets de 7 € ?

Si notre système possédait un billet ou une pièce de 7 €, l'algorithme glouton ne serait plus optimal. Prenons un montant de 14 €. L'algorithme glouton donne le rendu suivant 14=10+2+2 alors que le rendu optimal est 14=7×2.
Il existe des conditions sur le système pour que l'algorithme glouton soit optimal. Si le sujet vous intéresse, vous pouvez faire des recherches.

IV.3. Une solution optimale locale

Un algorithme glouton permet de trouver solution optimale locale mais pas toujours une solution optimale globale.

Reprenons notre exemple de rendu monnaie avec : S = $[1, 2, 20, 50, 100]$ et somme= 63.
L'algorithme glouton donne comme résultat : monnaie=$[50, 2, 2, 2, 2, 2, 2, 1]$

La solution optimale globale est pourtant : monnaie = $[20, 20, 20, 2, 1]$
Prenons maintenant l'exemple suivant : S=$[2, 5, 10, 20, 50, 100]$ et somme=21. L'algorithme glouton

ne va pas trouver de solution, alors qu'il existe au moins une solution locale : monnaie= $[5, 5, 5, 2, 2, 2]$

Exercice 2 :

1. Chercher une autre solution optimale locale meilleure que la précédente.
2. Faire une trace d'exécution de ces exemples.

Réponse :

> **Remarque :** Dans le système de pièces Européen, l'algorithme glouton donne toujours une solution optimale.

IV.4 Implémentation de l'algorithme glouton

Exercice 4 :

Programmer cet algorithme en langage python. Programmer une fonction rendu_monnaie_glouton(systeme,somme) qui possède comme paramètres

un système de pièces et de billets sous forme de liste de nombres ; une somme à rendre Cette fonction renvoie une liste de nombres qui caractérise la monnaie à rendre. Vous pouvez prendre comme jeu de test l'exemple de présentation. Il faudra penser à gérer le fait que l'algorithme ne trouve pas de solution dans certains cas.

Quelques aides : * penser à trier par ordre décroissant la liste des pièces et billets. * penser à gérer l'absence de solution globale.

> V EXERCICE **Objectifs** : - implémenter un algorithme glouton pour trouver une solution à un problème - implémenter l'algorithme de *force brute* pour déterminer la solution optimale et la comparer à la solution gloutonne ### Introduction 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 un poids 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 7 fichiers disponibles avec les durées données en minutes. | Nom | Durée en min (valeur) | Poids | |--------- |------ |--------- | | Vidéo A | 114 | 4.57 Go | | Vidéo B | 32 | 630 Mo | | Vidéo C | 20 | 1.65 Go | | Vidéo D | 4 | 85 Mo | | Vidéo E | 18 | 2,15 Go | | Vidéo F | 80 | 2,71 Go | | Vidéo G | 5 | 320 Mo | >**Problème** : Quelles vidéos copier sur la clé USB pour que la durée des vidéos soient la plus grande possible tout en ne dépassant pas 5 Go ? **Question 1** : Quelle est la valeur que l'on cherche à maximiser/minimiser ? Quelle est la contrainte ? Réponse : **Question 2** : Quel problème reconnaissez-vous ici ? Réponse : Vous allez implémenter (= programmer) une solution gloutonne pour résoudre ce problème, puis vous implémenterez l'algorithme de *force brute* qui donnera la solution optimale au

problème. La **stratégie gloutonne retenue sera de toujours prendre la vidéo de plus grande durée** n'excédant pas la capacité restante de poids. ### Représentation des données Dans la suite, on utilisera des dictionnaires de la forme suivante pour représenter chaque vidéo, la durée étant donnée minute et le poids étant donné en Go.

```
{'nom': 'Vidéo A', 'duree': 114, 'poids': 4.57}
```

On peut alors mémoriser les 7 vidéos dans le tableau `table_videos` suivant.

```
table_videos = [{'nom' : 'video A', 'duree' : 114, 'poids' : 4.57},
                {'nom' : 'video B', 'duree' : 32, 'poids' : 0.63},
                {'nom' : 'video C', 'duree' : 20, 'poids' : 1.65},
                {'nom' : 'video D', 'duree' : 4, 'poids' : 0.085},
                {'nom' : 'video E', 'duree' : 18, 'poids' : 2.15},
                {'nom' : 'video F', 'duree' : 80, 'poids' : 2.71},
                {'nom' : 'video G', 'duree' : 5, 'poids' : 0.32}]
```

On peut alors accéder aux éléments de cette table.

```
table_videos[0]['nom']
```

```
table_videos[3]['duree']
```

```
table_videos[6]['poids']
```

Ecriture des fonctions utiles #### Accès au poids d'une vidéo Pour résoudre le problème, on a besoin d'accéder au poids des vidéos pour vérifier la contrainte du poids maximal. La fonction suivante permet de renvoyer le poids d'une vidéo entrée en paramètre (sous la forme d'un dictionnaire comme vu précédemment). Quelques assertions devant être vérifiées par la fonction ont été écrites.

```
def poids(video):
    pass

assert poids({'nom' : 'video A', 'duree' : 114, 'poids' : 4.57}) == 4.57
assert poids({'nom' : 'video D', 'duree' : 4, 'poids' : 0.085}) == 0.085
assert poids({'nom' : 'video G', 'duree' : 5, 'poids' : 0.32}) == 0.32
```

Accès à la durée d'une vidéo De même, comme la stratégie gloutonne choisie prévoit de prendre la vidéo de plus grande durée (valeur) n'excédant pas la capacité restante de poids, il est nécessaire de pouvoir accéder aux durées de chacune des vidéos retenues. **Question 2** : Ecrivez une fonction `duree(video)` qui renvoie la durée (= valeur) d'une video entrée en paramètre (sous la forme d'un dictionnaire vu précédemment). Quelques assertions devant être vérifiées par votre fonction sont données ci-dessous.

```
def duree(video):
    # à compléter
```

```
assert duree({'nom' : 'video A', 'duree' : 114, 'poids' : 4.57}) == 114
assert duree({'nom' : 'video D', 'duree' : 4, 'poids' : 0.085}) == 4
assert duree({'nom' : 'video G', 'duree' : 5, 'poids' : 0.32}) == 5
```

Calculer le poids total et la durée totale d'une table de vidéos Enfin, il sera nécessaire de pouvoir calculer le poids et la durée totale d'une table de vidéos pour savoir si la contrainte de capacité maximale est respectée d'une part, et pour connaître la durée totale qui est la valeur à maximiser. **Question 3** : Ecrivez une fonction `poids_total(table_videos)` qui renvoie le poids total `poids_t` des vidéos présentes dans `table_videos`. Quelques assertions devant être vérifiées par votre fonction sont données ci-dessous.

```
def poids_total(table_videos):
    poids_t = 0
    # à compléter


    return poids_t
```

```
table1 = [{'nom' : 'video C', 'duree' : 20, 'poids' : 1.65}]
table2 = [{'nom' : 'video A', 'duree' : 114, 'poids' : 4.57},
          {'nom' : 'video B', 'duree' : 32, 'poids' : 0.63}]
table3 = [{'nom' : 'video A', 'duree' : 114, 'poids' : 4.57},
          {'nom' : 'video B', 'duree' : 32, 'poids' : 0.63},
          {'nom' : 'video C', 'duree' : 20, 'poids' : 1.65}]

assert poids_total(table1) == 1.65   # on compare des nombres réels mais on a
assert poids_total(table2) == 5.2
assert poids_total(table3) == 6.85
assert poids_total([]) == 0   # table vide testée
```

Question 4 : Ecrivez une fonction `duree_totale(table_videos)` qui renvoie la durée totale `duree_t` des vidéos présentes dans `table_videos`. Quelques assertions devant être vérifiées par votre fonction sont données ci-dessous.

```
def duree_totale(table_videos):
    duree_t = 0
    # à compléter


    return duree_t
```

```
table1 = [{'nom' : 'video C', 'duree' : 20, 'poids' : 1.65}]
table2 = [{'nom' : 'video A', 'duree' : 114, 'poids' : 4.57},
          {'nom' : 'video B', 'duree' : 32, 'poids' : 0.63}]
table3 = [{'nom' : 'video A', 'duree' : 114, 'poids' : 4.57},
          {'nom' : 'video B', 'duree' : 32, 'poids' : 0.63},
          {'nom' : 'video C', 'duree' : 20, 'poids' : 1.65}]

assert duree_totale(table1) == 20
assert duree_totale(table2) == 146
assert duree_totale(table3) == 166
assert duree_totale([]) == 0   # table vide testée
assert duree_totale(table_videos) == 273 # table avec toutes les vidéos
```

Implémentation de l'algorithme glouton #### Tri préalable Comme la stratégie gloutonne retenue prévoit de choisir les vidéos de plus grande durée n'excédant pas la capacité restante, il est judicieux de commencer par trier les vidéos de `table_videos` par durée décroissante. **Question 5** : Utilisez la fonction `sorted` de Python pour obtenir un *nouveau* tableau (appelé `table_triee`) contenant les vidéos de `table_videos` triées par durée décroissante. *Indication* : l'instruction `help(sorted)` permet d'afficher l'aide de la fonction `sorted` (vous pouvez aussi revoir le cours sur les tris de table si nécessaire).

```
# à compléter
table_triee = ...
table_triee # pour afficher le résultat
```

Algorithme glouton **Question 6** : en vous inspirant de l'implémentation de l'algorithme glouton du rendu de monnaie (voir cours), écrivez une fonction `glouton(table_videos, poids_max)` qui prend en paramètre une table de vidéos `table_videos` et un poids maximal `poids_max` et renvoie une liste `solution_gloutonne` contenant les vidéos de la solution gloutonne au problème. La solution gloutonne à trouver est donnée dans l'assertion qui suit.

```
def glouton(table_videos, poids_max):
    # TRI DE LA TABLE
    # à compléter par l'instruction de votre réponse précédente
    table_triee = ...
    # ALGORITHME GLOUTON
    poids_total = 0
    solution_gloutonne = []
    # à compléter
    # on se positionne sur la première vidéo (de la table triée)
    # tant qu'il reste des vidéos à traiter et que le poids max n'est pas at
        # on prend la premiere video restante
        # si elle n'est pas trop lourde (capacité restante suffisante)
            # on l'ajoute à solution_gloutonne et on met à jour le poids tota
        # on passe à la vidéo suivante

    return solution_gloutonne
```

```
assert glouton(table_videos, 5) == [{'nom': 'video A', 'duree': 114, 'poids'
                                    {'nom': 'video G', 'duree': 5, 'poids': (
                                    {'nom': 'video D', 'duree': 4, 'poids': (
```

Question 7 : Utilisez la fonction `duree_totale` pour déterminer la durée totale de la solution gloutonne. Quelle est sa durée totale ?

```
# à compléter
```

Implémentation de la *force brute* Le principe est simple : il faut étudier tous les cas possibles. Ainsi, pour appliquer cette stratégie, il faut : 1. d'abord *énumérer* toutes les combinaisons possibles de vidéos 2. puis conserver celles dont la capacité maximale n'est pas dépassée 3. enfin, trouver la meilleure solution parmi les combinaisons restantes >**Remarque** : Dans un souci de simplicité, il a été fait le choix de ne pas utiliser de fonctions pour implémenter la stratégie de force brute. Vous pourrez trouver un code plus propre (mais plus complexe) en suivant ce [lien](#).

```
table_videos = [{'nom' : 'video A', 'duree' : 114, 'poids' : 4.57},
                {'nom' : 'video B', 'duree' : 32, 'poids' : 0.63},
                {'nom' : 'video C', 'duree' : 20, 'poids' : 1.65},
                {'nom' : 'video D', 'duree' : 4, 'poids' : 0.085},
                {'nom' : 'video E', 'duree' : 18, 'poids' : 2.15},
                {'nom' : 'video F', 'duree' : 80, 'poids' : 2.71},
                {'nom' : 'video G', 'duree' : 5, 'poids' : 0.32}]
```

Etape 1 : énumération de toutes les combinaisons Il s'agit de la difficulté majeure. Dans une combinaison de vidéos, chaque vidéo de départ est prise ou non, il s'agit d'une donnée binaire. Ainsi, une approche consiste à créer des mots binaires représentant chaque combinaison. Par exemple, le mot '1101001' signifie qu'on prend les vidéos A, B, D et G tandis que le mot '1111111' signifie que l'on prend toutes les vidéos. Dans notre exemple, nous avons 7 vidéos donc il y a $2^{7} = 128$ combinaisons possibles. De manière générale il y a donc 2^n combinaisons pour un ensemble de n vidéos. On va construire un tableau `combinaisons` contenant toutes les combinaisons binaires. Pour cela, l'idée est de commencer par construire un tableau `tab_entiers` contenant tous les entiers compris entre 0 et 2^n-1.

```
n = len(table_videos)  # nombre de vidéos
tab_entiers = [i for i in range(2**n)]  # création d'un tableau avec tous les
print(tab_entiers)
```

Ensuite, on construit un autre tableau `tab_binaire` avec les conversions binaires de chaque entier. >**Astuce** : la fonction `bin` prend un entier en paramètre et renvoie sa valeur binaire sous forme d'une chaîne de caractères. Par exemple, `bin(12)` renvoie la chaîne `'0b1101'`. Il suffira de supprimer les caractères `'0b'` en tête pour obtenir l'écriture binaire. On veillera également à compléter avec des zéros devant pour obtenir un mot de la longueur désirée.

```
tab_binaire = [bin(i)[2:] for i in tab_entiers]  # conversion binaire des en
print(tab_binaire)
```

Enfin, pour obtenir le tableau `combinaisons`, on prendra le soin de compléter avec autant de zéros que nécessaires les valeurs binaires précédentes pour obtenir des mots de longueur 7, représentant les combinaisons possibles de vidéos.

```
combinaisons = ['0'*(n-len(k)) + k for k in tab_binaire]  # ajout des zéros
```

On peut vérifier que le tableau `combinaisons` contient bien tous les 128 mots binaires de longueur 7.

```
combinaisons
```

Etape 2 : conservation des combinaisons valides On cherche maintenant à conserver uniquement les combinaisons valides, c'est-à-dire celles ne dépassant pas la capacité maximale de 5 Go. Par exemple : - `'0000001'` est à conserver puisqu'il s'agit uniquement de la vidéo G de 0.32 Go - `'1010000'` n'est pas à conserver car le poids total de cette combinaison (vidéos A et C) vaut 4.57 + 1.65 = 6.22 Go, ce qui dépasse la

capacité maximale autorisée. Pour cela, on peut calculer le poids total de chaque combinaison. Si celui-ci est inférieur au poids maximal autorisé, alors la combinaison est valide et on la conserve. **Question 8** : Complétez le programme suivant pour qu'en fin d'exécution la liste `combinaisons_valides` contienne tous les couples `(combi, duree_combi)` des combinaisons valides et de leur durée. L'assertion qui suit indique le nombre de combinaisons valides.

```
n = len(table_videos)
poids_max = 5
combinaisons_valides = []
for combi in combinaisons: # on parcourt chaque combinaison du tableau combi
    poids_combi = 0
    duree_combi = 0
    for i in range(n):  # on parcourt la combinaison caractère par caractère
        # à compléter
        # si le caractère est '1', alors on met à jour la durée et le poids
    # si la combi est valide alors on ajoute le couple (combi, duree_combi)à
```

```
assert len(combinaisons_valides) == 51
```

Si vous ne parvenez pas à trouver la liste souhaitée, la voici. Vous pourrez continuer en l'utilisant.

```
combinaisons_valides = [('0000000', 0),
                        ('0000001', 5),
                        ('0000010', 80),
                        ('0000011', 85),
                        ('0000100', 18),
                        ('0000101', 23),
                        ('0000110', 98),
                        ('0001000', 4),
                        ('0001001', 9),
                        ('0001010', 84),
                        ('0001011', 89),
                        ('0001100', 22),
                        ('0001101', 27),
                        ('0001110', 102),
                        ('0010000', 20),
                        ('0010001', 25),
                        ('0010010', 100),
                        ('0010011', 105),
                        ('0010100', 38),
                        ('0010101', 43),
                        ('0011000', 24),
                        ('0011001', 29),
                        ('0011010', 104),
                        ('0011011', 109),
                        ('0011100', 42),
                        ('0011101', 47),
                        ('0100000', 32),
                        ('0100001', 37),
                        ('0100010', 112),
                        ('0100011', 117),
                        ('0100100', 50),
                        ('0100101', 55),
                        ('0101000', 36),
                        ('0101001', 41),
                        ('0101010', 116),
```

```
                          ('0101011', 121),
                          ('0101100', 54),
                          ('0101101', 59),
                          ('0110000', 52),
                          ('0110001', 57),
                          ('0110010', 132),
                          ('0110100', 70),
                          ('0110101', 75),
                          ('0111000', 56),
                          ('0111001', 61),
                          ('0111100', 74),
                          ('0111101', 79),
                          ('1000000', 114),
                          ('1000001', 119),
                          ('1001000', 118),
                          ('1001001', 123)]
```

Etape 3 : trouver la meilleure solution C'est très simple puisqu'il suffit de déterminer la combinaison valide de durée maximale. Cela revient donc à une recherche de maximum. **Question 9** : Ecrivez un programme permettant de trouver la solution optimale et sa durée totale.

```
# à compléter
```

Vous devez trouver que la solution optimale est la combinaison `'0110010'` d'une durée de 132 min. **Question 10** : Comparez la solution optimale et la solution gloutonne trouvée précédemment. Réponse : **Question 11** : Ecrivez un programme permettant d'afficher une liste `liste_optimale` contenant les noms des vidéos correspondant à la solution optimale.

```
liste_optimale = []
# à compléter


print("la meilleure solution est de choisir :", liste_optimale)
```