

Algorithmes gloutons

La méthode glouton est une méthode de recherche de solution locale qui va apporter **une solution globale non nécessairement optimale**. Le problème se ramène à chaque étape à un problème plus simple et **chaque étape sélectionne l'une des meilleures possibilités et ne remet jamais en cause les choix précédents**.

Parmi les exemples classiques, le rendu de monnaie: on veut rendre la monnaie de façon optimale c'est à dire avec le moins de pièces possible.

Quelle est la stratégie la plus simple à adopter? Rendre la plus grande pièce possible à chaque étape ? Par exemple, en euros, si je dois rendre 195 ct, je rends en premier 1€ (100ct), puis 50 ct puis 20ct puis 20ct puis 5ct. Cette stratégie fonctionne avec les euros. On peut par contre imaginer un jeu de pièces pour lequel la stratégie glouton n'est pas optimale (par exemple 3 ; 2 ; 0.5 . Si vous voulez rendre 4 euros, cette stratégie vous fera rendre 3 pièces alors qu'il aurait été possible de n'en rendre que 2), voire un jeu de pièces où la méthode glouton ne fonctionnerait pas du tout.

Autrement dit, dans certains cas, la méthode Glouton ne trouve pas de solution alors qu'il en existe une. Pour d'autres problèmes, la méthode n'est pas optimale car elle donne bien une solution mais pas la meilleure. Enfin, pour certains problèmes, la méthode glouton est optimale. Lorsqu'on choisit d'utiliser cette méthode, il convient d'y prêter attention.

Voilà un code de rendu de monnaie (*rendu-monnaie.py* disponible en ligne):

```
def rendu_glouton(pieces,montant):
    reponse=[]
    i=len(pieces)-1
    while i>=0 and montant>0 :
        if montant<pieces[i]:
            i=i-1
        else:
            montant=montant-pieces[i]
            reponse.append(pieces[i])
    if i<0:
        reponse=[]
    return(reponse)
```

- ➔ Faites fonctionner dans votre IDE préféré ce code (*pieces* est la liste contenant le jeux de pièces disponible, en cts d'euros ; *montant* le montant de la monnaie à rendre)
- ➔ S'agit-il d'un algorithme glouton ? (justifiez par l'ajout de commentaires. En cas de difficultés à répondre à cette question, n'hésitez pas à le tourner à la main (cf doc adhoc)

Le problème du sac à dos :

Un cambrioleur possède un sac à dos d'une contenance maximum de 30 Kg. Au cours d'un de ses cambriolages, il a la possibilité de dérober 4 objets A, B, C et D. Voici un tableau qui résume les caractéristiques de ces objets :

objet	A	B	C	D
masse	13 Kg	12 Kg	8 Kg	10 Kg
valeur marchande	700 €	400 €	300 €	300 €
valeur au poids	54 €/Kg	33 €/Kg	38 €/Kg	30 €/Kg

Déterminez les objets que le cambrioleur aura intérêt à dérober, sachant que :

- tous les objets dérobés devront tenir dans le sac à dos (30 Kg maxi)
- le cambrioleur cherche à obtenir un gain maximum.

➔ Analysez la démarche que vous avez utilisée et demandez-vous si elle donne une solution optimale.

1ere méthode : **on essaie toutes les possibilités** et on voit celle qui est optimale. Normalement, on est nombreux dans la classe à avoir utilisé cette méthode de « **force brute** », en mettant de côté les solutions qui nous paraissaient « instinctivement » pas bonne (genre prendre un seul objet). Notons au passage que la machine est dénuée d'instinct et que donc, si je veux utiliser cette méthode, je serai obligé de faire tester toutes les combinaisons.

Ici, avec 4 objets, il y a 16 tests à faire (2 puissance 4). Si on ajoute un objet (5 au lieu de 4), on passe à 32 tests, 64 tests pour 6 objets. **Pour 50 objets (ce qui n'est pas énorme), on passe à environ 100 000 000 000 000 tests !** Le nombre de test croit exponentiellement. Cet algorithme, qui certes nous donnera à coup sûr une solution optimale, a donc une **complexité exponentielle** ($O(2^n)$), ce qui, rappelons nous, est exécrable. Il n'est donc pas possible d'utiliser cette méthode lorsque les objets sont nombreux si on souhaite un résultat avant la fin du siècle (et un cambrioleur a rarement plusieurs siècles devant lui pour commettre son méfait....).

Il sera donc très souvent préférable d'utiliser un algorithme glouton qui ne nous donnera pas à coup sûr la solution optimale mais quelque chose de « pas trop mal » dans un temps raisonnable.

- Si je fais mon choix avec la **méthode de complexité exponentielle**, je prendrais l'objet A + l'objet B, ce qui me fera un total de 1100 euros, ce qui est la **solution optimale**.
- Si je fais mon choix en utilisant une **méthode gloutonne** avec la ligne « valeur au poids », je prendrais l'objet A + l'objet C, ce qui me fera un total de 1000 euros, **ce qui n'est pas optimal**.
- Si je fais mon choix en utilisant une **méthode gloutonne** avec la ligne « valeur marchande », je prendrais l'objet A + l'objet B, ce qui me fera un total de 1100 euros, **ce qui, coup de chance est la solution optimale**.
-

Il est important de bien comprendre qu'un algorithme glouton ne donne pas forcément une solution optimale.

Expérimentation : construire un planning de festivalier.

Notre problème :

Vous vous rendez à un festival de rap. Ce festival a plusieurs scènes. Vous souhaitez voir le plus de groupes possible. Vous ne faites pas vos choix en fonction du groupe (ils font tous exactement la même chose, c'est un festival de rap (je sens que je vais en faire hurler qqes uns 😊)), juste vous voulez revenir du festival et pouvoir dire « j'ai vu plein de groupes » et vous la jouer en donnant le plus grand nombre possible.

Vous devez voir les sets des groupes dans leur entier (interdit de rater le début ou la fin d'un set, le service d'ordre est pénible et implacable).

Voilà la prog de ce festival:

Lorenz.s.b.d.m.		naska		section du lassot		nekeau		Snoop Chiwawaa		dam pas si sot		padawan pims								
boobi	djoule		tuordemavu		Supreme N.T.NSI				50 euros		PPNL	grand oli et fléau								
19h	19h30		20h		20h30		21h		21h30		22h		22h30		23h		23h30		24h	

On a regroupé dans une **liste de tuple** les horaires (heure de début, heure de fin , nom de l' « artiste »)

```
prog = [  
( 20 , 21 , "section du lassot" ) ,  
( 20.5 , 22 , "Supreme N.T.NSI" ) ,  
( 19 , 19.25 , "boobi" ) ,  
( 19.25 , 19.75 , "djoule" ) ,  
( 19.5 , 20 , "naska" ) ,  
( 21 , 21.5 , "nekeau" ) ,  
( 22.5 , 22.75 , "PPNL" ) ,  
( 19.75 , 20.5 , "tuordemavu" ) ,  
( 22.5 , 23 , "dam pas si sot" ) ,  
( 22.75 , 24 , "grand oli et fléau" ) ,  
( 23 , 24 , "padawan pims" ) ,  
( 19 , 19.5 , "Lorenz.s.b.d.m." ) ,  
( 21.5 , 22.5 , "Snoop Chiwawa" ) ,  
( 22 , 22.5 , "50 euros" )  
]
```

Nous allons tenter d'écrire un algorithme glouton qui guide vos choix puis de l'implanter en Python.

N'oublions pas le principe d'un algo glouton : Chaque choix fait sélectionne l'une des meilleures possibilités et ne remet jamais en cause les choix précédents.

Une première solution gloutonne est de choisir les concerts les plus courts. Problème : Ils sont plusieurs à être le « plus court » ! Il faut donc donner un critère de choix à l'algorithme. On peut choisir de **prendre le premier plus court (celui qui commence le plus tôt)**, puis recommencer l'opération en veillant à ce que son créneau n'empiète pas sur le créneau déjà sélectionné. En résumé : après avoir classé les intervalles par valeurs croissantes des heures de début, on va sélectionner l'intervalle de la première plus petite durée, puis celui de la deuxième plus petite valeur compatible avec la précédente, et ainsi de suite.

Une seconde solution gloutonne :

- Classer les intervalles par **heures de fin croissantes**.
- Choisir le groupe associé au premier intervalle.
- Choisir parmi les intervalles suivants celui du concert dont l'intervalle est compatible avec celui du premier concert.
- Recommencer ainsi avec les intervalles classés suivants jusqu'à ce qu'il n'y en ait plus à traiter.

Cette solution semble sympathique et plus simple à mettre en œuvre que la précédente.

Il nous faut donc coder :

- Une fonction qui trie le dictionnaire par heures de fin croissantes. On l'appellera *classement_horaires()*
- Une fonction qui cherche dans la liste triée le concert qui se termine le plus tôt
- Une fonction gloutonne qui fabrique notre planning. On l'appellera *planning()*

1/ tri

On connaît déjà deux algorithmes de tri (par sélection ou par insertion). Choisissons n'importe lequel des deux, en tri sur place ou non, peu importe, à notre convenance.

➔ Reconnaître si le code ci-dessous code un algorithme de tri par insertion ou par sélection.

```
def classement_horaires(prog) :  
    j=1  
    while j < len(prog):  
        i=j-1  
        groupe_et_horaires=prog[j] #groupe à trier dans l'iteration  
        heure_fin=groupe_et_horaires[1] # ou k2 = prog[j][1] , heure de fin du groupe à trier  
        while (heure_fin<prog[i][1] and i>=0): # prog[i][1] , c'est l'heure de fin du groupe comparé au groupe à trier  
            prog[i+1]=prog[i]  
            i=i-1  
        prog[i+1]=groupe_et_horaires  
        j=j+1  
    return(prog)
```

Tri par :

➔ Faites le tourner dans votre IDE préféré pour voir si cela fonctionne.
(vous avez le fichier *tri-par-horaire-de-fin.py* de dispo)

Nous voilà donc avec notre liste de tuples triée.

2/ recherche du concert qui termine le plus tôt :

On sait faire un algorithme qui recherche la plus petite valeur d'une liste (cf cours « *Rappels avant de faire les algo de tri.pdf* »).

➔ Ecrire un algorithme d'une fonction qui recherche ce concert. Elle prendra en paramètre d'entrée la liste triée et l'heure à partir de laquelle il faut chercher. Elle renverra une liste contenant l'index de ce concert dans la liste triée ainsi qu'un booléen indiquant le succès ou non de cette recherche (on appellera cette liste *infos*. Implantez en Python cette fonction que vous nommerez *trouver_index_du_concert_qui_fini_le_plus_tot_apres_le_precedent()*.

3/ gloutonnerie

Il est temps d'utiliser notre algorithme glouton pour établir notre planning. Notre programme doit :

- Trier la prog par horaire de fin de concert
- Choisir le concert qui fini le plus tôt et l'insérer en tête de notre liste planning
- Choisir le concert qui fini le plus tôt après le dernier de notre planning et l'insérer dans le planning
- Répéter l'opération précédente jusqu'à ce qu'il n'y ai plus de solution

On a déjà à dispo deux fonctions :

- **classement_horaires()** qui nous trie la prog par horaire de fin
- **trouver_index_du_concert_qui_fini_le_plus_tot_apres_le_precedent()** qui nous trouve l'index du concert suivant et nous dit si il a échoué ou non.

Ecrivez cet algorithme puis implantez-le en Python :

```
VAR prog en liste de tuple  
VAR planning en liste de tuple  
VAR h en numérique  
VAR pas_fini en booléen ( drapeau qui nous dit si le travail de planning est terminé ou non)
```

Expliquez pourquoi l'algorithme que nous avons utilisé est bel et bien un algorithme glouton :

Pour aller plus loin sur cet exercice :

On utilise maintenant une liste de la prog où figure un quatrième élément dans les tuples : une note sur 20 attribuée en fonction de vos goûts. On change le critère de choix. Vous ne souhaitez plus en voir le plus possible mais les meilleurs artistes. Le but est donc d'engranger le plus de points.

Proposez un programme qui vous fait votre planning

Conclusion :

UN ALGORITHME GROUTON EST UN ALGORITHME QUI CHOISIT CE QUI LUI SEMBLE ETRE LA MEILLEURE SOLUTION A CHAQUE ETAPE SANS JAMAIS REVENIR SUR SA DECISION.

UN ALGORITHME GROUTON NE DONNERA PAS FORCEMENT LA SOLUTION OPTIMALE.

UN ALGORITHME GROUTON A UNE COMPLEXITE BIEN MOINDRE QU'UNE METHODE DE FORCE BRUTE (QUI EST EXPONENTIELLE ET DONC RAPIDEMENT REDHIBITOIRE)

C'est un des algorithmes de base en informatique.