Lycée Jean Moulin **Draguignan**
NSI 1re Année .....

Les algorithmes gloutons

Le principe

On cherche une solution optimale à un problème.

La technique la plus basique pour résoudre un problème consiste à énumérer de façon exhaustive toutes les solutions possibles, puis à choisir la meilleure.

Cette approche par force brute, toujours possible, n'est sans doute pas optimale...

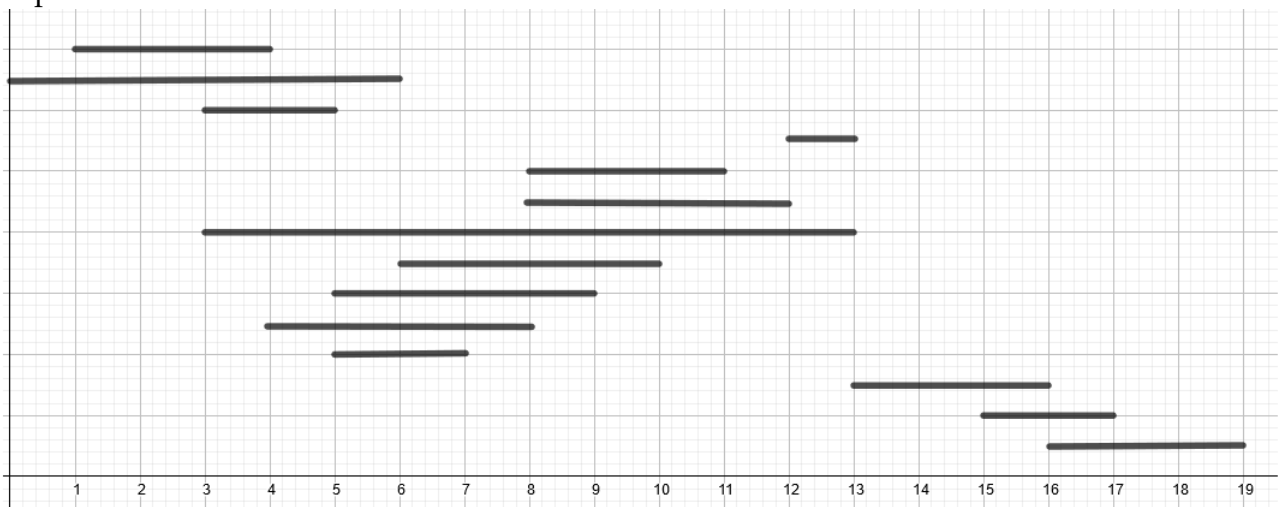
La méthode gloutonne consiste à choisir des solutions locales optimales d'un problème dans le but d'obtenir une solution optimale globale au problème.

Je comprend par l'exemple

Le problème du choix d'activités:
On considère un ensemble d'activités, chacune possédant **une date de début** et **une date de fin**. Pour des raisons logistiques, on ne peut pas programmer des activités se déroulant **en même temps**.
La question est la suivante : quel est **le nombre maximal d'activités** que l'on va pouvoir planifier ?

Prenons un exemple pour se fixer les idées, les activités sont représentées par une liste de tuples (heure_début , heure_fin).
Activités = [(1,4),(0,6),(3,5),(12,13),(8,11),(8,12),(3,13),(6,10),(5,9),(4,8),(5,7),(13,16),(15,17),(16,19)]

Résolution du problème à la main
Représentons ces activités sur un schéma :



À faire 1:
Vérifier que le nombre maximal d'activités que l'on peut planifier est 6.
Y a-t-il une unique solution ?

Pour des situations plus complexes, un schéma ne nous aideras pas à trouver une solution, nous devons faire appel à un algorithme.

1ère approche : Tri par durée.
Une première méthode consiste à prendre les activités par ordre croissant de leur durée.
On peut en effet raisonnablement penser que planifier d'abord les activités les plus courtes laissera plus de temps pour les autres.

À FAIRE 2:
Trier la liste par ordre croissant de leur durée.
.....
.....
Quelle solution obtient-on ? Est-elle optimale ?
.....
.....
.....
.....

2nde approche : Tri par date de début.
Considérons les activités par ordre croissant de leur date de début.

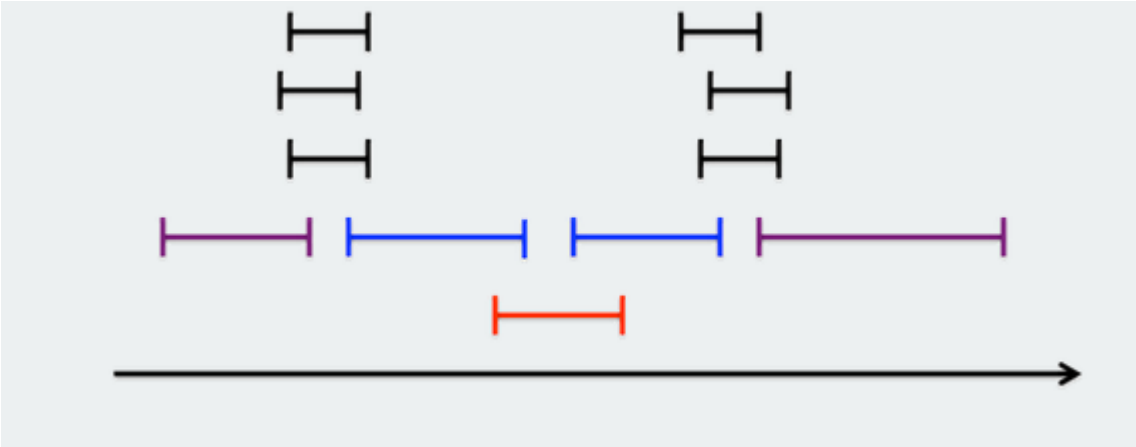
À FAIRE 3:
Trier la liste par ordre croissant de leur début.
.....
.....
Quelle solution obtient-on ? Est-elle optimale ?
.....
.....
.....
.....

3ème approche : Tri par nombre d'incompatibilités.
Considérons les activités par ordre croissant de leurs incompatibilités.
Procéder ainsi permet de privilégier les activités qui "gênent" le moins, et donc avoir moins de contraintes pour planifier les autres.

À FAIRE 4:
Compléter le tri suivant, où (12,13,1) représente la plage horaire (12,13) qui a 1 incompatibilité.
[(12,13,1),(13,16,1), ...
.....

On obtient l'une des solutions optimales : [(12,13,1),(13,16,1),(16,19,1),(1,4,3),(8,11,4),(5,7,5)]

La solution obtenue n'est cependant pas optimale dans tous les cas.
En effet, avec cette configuration, cette méthode aurait donné 3 activités au lieu des quatre possibles...



La bonne idée : Tri par date de fin.
On considère les activités par ordre croissant de leur date de fin.
On va ainsi chercher à laisser le plus de temps disponible après la planification d'une activité.

À FAIRE 5:
Trier les activités par ordre croissant de leur date de fin.
.....
.....
Donner toutes les solutions possibles
.....
.....
.....

Pour les accros

Réaliser un programme qui propose une planification optimale pour ces activités.

D'autres exemples

Le problème du rendu de monnaie.
Lorsque vous passez à la caisse d'un magasin quelconque, il n'est pas rare que le caissier doive vous rendre de l'argent car le montant que vous lui avez donné est supérieur à celui que vous devez payer.
Supposons qu'on doive vous rendre la somme de 2,63€.
Il y a plusieurs façons de procéder. On peut par exemple vous rendre 263 pièces de 1 cent, 125 pièces de 2 cents et 13 pièces de 1 cent ou encore 5 pièces de 50 cents, une de 10 cents, une de 2 cents et enfin une de 1 cent.
Il y a bien évidemment énormément de possibilités pour vous rendre la dite somme.
Il y a fort à parier que les solutions du type "263 pièces de 1 cent" ne vous conviennent pas, pour cause, personne n'a envie de remplir son porte monnaie avec autant de pièces...

Le problème qui se pose est donc de minimiser le nombre de pièces rendues pour un montant fixé.

À faire 6:

Proposer un algorithme qui résout ce problème.
On considérera la somme à rendre en centimes et la liste des pièces disponibles :
[1,2,5,10,20,50,100,200]
Implémenter l'algorithme en Python.

Question 1:

imaginons un système monétaire dans lequel il y aurait seulement des pièces de 1, 3 et 4 unités.
Supposons qu'on doive me rendre 6 unités monétaires.
Que produit votre algorithme, est-il optimal dans ce cas ?

.....
.....
.....
.....
.....
.....

Le problème du sac à dos.

On dispose d'un ensemble d'objets.
Chaque objet possède une valeur b et un poids W.
On souhaiterait prendre une partie T de ces objets dans notre sac-à-dos, malheureusement, ce dernier dispose d'une capacité limitée W. On ne pourra pas toujours mettre tous les objets dans le sac étant donné que la somme des poids des objets ne peut pas dépasser la capacité maximale.
On va cependant chercher à maximiser la somme des valeurs des objets qu'on va emporter avec soi.

À faire 7:

L'idée à suivre, si on veut développer une méthode gloutonne, est d'ajouter les objets de valeurs élevées en premier, jusqu'à saturation du sac.
Cette méthode est parfois efficace, mais parfois pas.
Supposons qu'on dispose d'un sac de capacité W = 26 kg et des objets suivants :

Objets	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Valeurs	4	3	8	5	10	7	1	7	3	3	6	12	2	4
Poids	2	2	5	2	7	4	1	4	2	1	4	10	2	1

Proposer un algorithme qui résout ce problème.
Écrire le programme correspondant.

Question 2:

Qu'en est-il de votre algorithme dans ce cas:

Objets	A	B	C	D	E	F
Valeurs	30	12	12	12	12	4
Poids	39	10	10	10	10	1

.....
.....

Comment faire pour avoir une solution optimale?
Il n'y a pas de réponse miracle à cette question, tout dépend du problème.
Dans certains cas, les algorithmes gloutons produisent d'excellents résultats et sont appropriés au problème, dans d'autres cas, non.
Généralement, si les poids des objets sont très déséquilibrés, les algorithmes gloutons produiront une solution non optimale car de tels algorithmes ont une mauvaise vision globale du problème.
Il faut réfléchir à la solution à adopter en fonction du problème.
Pour un projet, trouver d'autres exemples que l'on peut traiter avec une méthode gloutonne...

Un exercice : Les stations services

Vous souhaitez vous rendre de Liège à Brest en scooter.
Votre réservoir vous permet de rouler R Km.
Vous connaissez la liste des pompes à essence disponibles sur la route, donnée sous la forme d'une liste S = $[d_1,d_2,...,d_k]$ où chaque d_i donne la distance qui le sépare de son précédent.

S[0] à d_1 kilomètres du départ.
S[1] à d_2 kilomètres de $s[d_1]$.
Etc.

On suppose $d_i \leq$ R pour $i = 1..k$, d_k symbolisant l'arrivée.
On souhaite faire le moins d'arrêt possible.

À FAIRE 8:

Écrire un algorithme glouton en pseudo-code résolvant le problème, c'est à dire renvoyant la liste des pompes à essence où l'on doit s'arrêter.
Donner une version en python de votre algorithme.
Pour un réservoir de 250 Km, tester avec la liste [120,142,90,70,130,150,84,25,110].

Algorithme:

Programme

```
_
```

Corrigés

À faire 2 :

[(12,13),(3,5),(5,7),(15,17),(1,4),(8,11),(13,16),(16,19),(8,12),(6,10),(5,9),(4,8),(0,6),(3,13)]
solution: [(12,13),(3,5),(5,7),(15,17),(8,11)] , non

À faire 3 :

[(0,6),(1,4),(3,5),(3,13),(4,8),(5,9),(5,7),(6,10),(8,11),(8,12),(12,13),(13,16),(15,17),(16,19)]
solution: [(0,6),(6,10),(12,13),(13,16),(16,19)] , non

À faire 4 :

[(12,13,1),(13,16,1),(16,19,1),(15,17,2),(1,4,3),(3,5,4),(8,11,4),(8,12,4),(5,7,5),(0,6,6),(6,10,6),(4,8,6),(5,9,7
solution: [(12,13,1),(13,16,1),(16,19,1),(1,4,3),(8,11,4),(5,7,5)] , non

À faire 5 :

[(1,4),(3,5),(0,6),(5,7),(4,8),(5,9),(6,10),(8,11),(8,12),(12,13),(3,13),(13,16),(15,17),(16,19)]
solutions: [(1,4),(4,8),(8,12),(12,13),(13,16),(16,19)]
[(1,4),(4,8),(8,11),(12,13),(13,16),(16,19)]
[(1,4),(5,7),(8,11),(12,13),(13,16),(16,19)]
[(1,4),(5,7),(8,11),(12,13),(13,16),(16,19)] , oui

```
def Tri_date_fin(S):
    '''
    Programme qui donne un planning
    on commence par la première
    la suivante sera la première compatible
    et ainsi de suite
    '''
    S.sort(key=lambda x:x[1])
    C=[]
    C.append(S[0])
    j=0
    for i in range(j+1,len(S),1):
        if S[i][0]>=S[j][1]:
            C.append(S[i])
            j=i
    return C

S = [(1,4),(0,6),(3,5),(12,13),(8,11),(8,12),
(3,13),(6,10),(5,9),(4,8),(5,7),(13,16),(15,17),(16,19)]

sol=Tri_date_fin(S)
print(sol)
```

```
# rendu de monnaie
liste1=[0,0,0,0,0,0,0,0]
liste=[200,100,50,20,10,5,2,1]
def monnaie(s):
    for i in range(0,8):
        liste1[i]=s//liste[i]
        s=s%liste[i]

somme=int(input('Entrez le montant de la monnaie en centimes'))
print(somme)
monnaie(somme)
print(liste1)
```

```
''' Ici on utilise un dictionnaire pour représenter
    les données. On peut le faire avec d'autres représentations
'''
# algo du knapsack
mon_dico={'A':(4,2),'B':(3,2),'C':(8,5),'D':(5,2),'E':(10,7),'F':(7,4),
'G':(1,1),'H':(7,4),'I':(3,2),'J':(3,1),'K':(6,4),'L':(12,10),
'M':(2,2),'N':(4,1)}

#tri du dictionnaire par valeur 1 du tuple -->liste
def get_val(mon_dico):
    return mon_dico[1][0]
liste=sorted(mon_dico.items(),reverse=True,key=get_val)
# liste est une liste qui contient tuple(clé,tuple(valeur,poids))

W_max=26
w=0
print(liste)
liste1=[]
i=0
while i<len(liste):
    if w<=W_max:
        w=w+liste[i][1][1]
        if w>W_max:
            break
        liste1.append(liste[i][0])
    i=i+1
s=0
for i in range(0,len(liste1)):
    s=s+liste[i][1][1]

print("le sac contient les objets : ",liste1," pour un poids de :",s,"kg")
```

Données : L = [120,142,90,70,130,150,84,25,110]
st une liste vide
$s \leftarrow 0$
$i \leftarrow 0$
$n \leftarrow$ taille de L
Tant que i < n et s < 906 **faire**
    s ← s + L[i]
    Si s > = 250 **alors**
        $st \leftarrow$ L[i-1]
        $s \leftarrow 0$
        $i \leftarrow i - 1$
    $i \leftarrow i + 1$
Afficher : st

```
st=[]
L=[120,142,90,70,130,150,84,25,110]
s=0
i=0
while i < len(L) and s<906:
    s=s+L[i]
    if s>250:
        st.append(L[i-1])
        s=0
        i=i-1
    i=i+1
print(st)
```