Les algos Gloutons

ou comment résoudre un problème en maximisant son gain ?

Algorithmie "Classique"

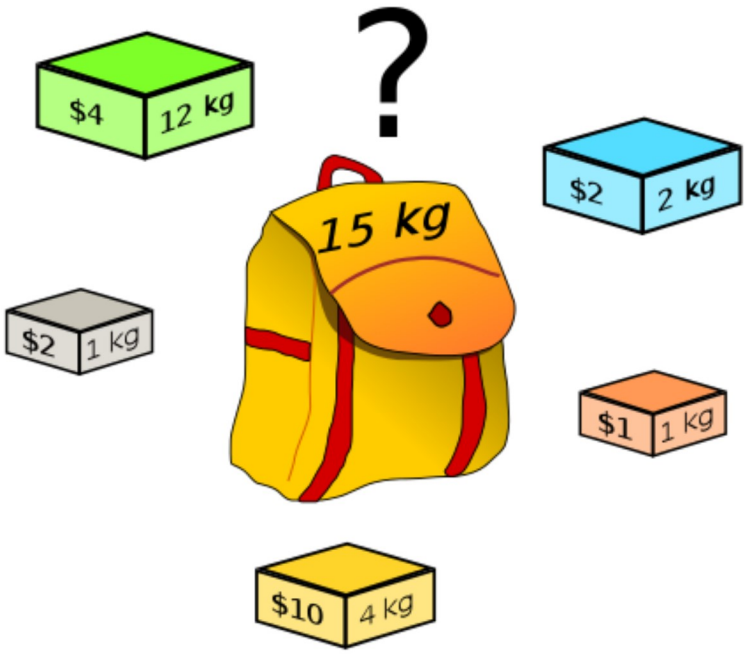
Il existe dans la littérature informatique deux problèmes classiques, dont les informaticiens ont essayé de généraliser la résolution.

Le rendu de monnaie

Le problème du rendu de monnaie s'énonce de façon simple : étant donné un ensemble de pièces à disposition (je ne peux rendre que des pièces de 50 centimes, 1 euro, 2 euros…) et un montant à rendre, rendre ce montant avec un nombre minimal de pièces du système que l'on s'est donné. Les applications d'une solution à ce problème sont faciles à imaginer : nul n'a envie de récupérer 1 euro en pièces de 1 centime s'il s'est aventuré à payer 2 euros pour une malheureuse bouteille de soda à un distributeur !!

Lien vers l'article de Wikipedia (https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_rendu_de_monnaie)

le sac à dos



Lien vers l'article de Wikipedia (https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_sac_%C3%A0_dos) :

En algorithmique, le problème du sac à dos, noté également KP (en anglais, Knapsack problem) est un problème d'optimisation combinatoire. ... Ou plus simplement, les objets mis dans le sac à dos doivent maximiser la valeur totale, sans dépasser le poids maximum.

Pour le cours de NSI, j'ai choisi des problèmes de types "sac à dos". C'est un choix personnel. Le programme de NSI laisse le choix entre les 2 types de probléme.

Algorithmes gloutons	Résoudre un problème grâce à un algorithme glouton.	Exemples : problèmes du sac à dos ou du rendu de monnaie. Les algorithmes gloutons constituent une méthode algorithmique parmi d'autres qui seront vues en terminale.

La NASA

Vous faites partie de l'équipage d'un vaisseau spatial qui doit rejoindre une base installée sur la face visible de la Lune. Mais, suite à un certain nombre de problèmes, vous êtes contraints de vous poser en catastrophe, à 320 km du point prévu.

Au cours de cet alunissage d'urgence, la plupart des équipements de bord de votre vaisseau sont endommagés, à l'exclusion de vos scaphandres de sortie dans l'espace et de quinze objets cités ci-dessous.

Il s'agit donc pour votre équipage de rejoindre la base lunaire au plus vite en emportant le strict minimum avec vous. Vous ne pouvez emmener que 10 objets parmi les 15 cités ci -dessous.

- Une boîte d’allumettes
- Des aliments concentrés
- 50 mètres de corde en nylon
- Un parachute en soie
- Un appareil de chauffage fonctionnant sur l’énergie solaire
- Deux pistolets calibre 45
- Une caisse de lait en poudre
- Deux réservoirs de 50 kg d’oxygène chacun
- Une carte céleste des constellations lunaires
- Un canot de sauvetage autogonflable
- Un compas magnétique

Quelle(s) structure(s) algorithmique(s), vous paraissent pertinentes pour enregistrer vos valeurs ?

- Codez l'une de vos solutions.
- Puis trier les par ordre décroissant de leur "valeur" (vous pouvez utiliser la fonction `sort` de Python).

In [4]:
```
#*complétez ici votre sructure de données.*
# J'ai choisi ici une liste de dictionnaire.
#Chaque dictionnaire représente un des objet NASA, pour chacun d'entre eux,
#on note dans 'objet' le nom de l'objet et dans 'valeur' sa valeur attribuée.
liste=[{'objet': 'allumettes', 'valeur': '01'}, {'objet': 'aliments', 'valeur': '12
'},
 {'objet': 'corde', 'valeur': '10'},
 {'objet': 'parachute', 'valeur': '08'},
 {'objet': 'chauffage', 'valeur': '03'},
 {'objet': 'pistolets', 'valeur': '05'},
 {'objet': 'lait', 'valeur': '04'},
 {'objet': 'oxygène', 'valeur': '15'},
 {'objet': 'carte', 'valeur': '13'},
 {'objet': 'canot', 'valeur': '07'},
 {'objet': 'compas', 'valeur': '02'},
 {'objet': 'eau', 'valeur': '14'},
 {'objet': 'seringues', 'valeur': '09'},
 {'objet': 'signaux', 'valeur': '06'},
 {'objet': 'emetteur', 'valeur': '11'}]
#*Trier votre structure par ordre décroissant de la valeur
liste_ordo =sorted(liste, key=lambda d: d["valeur"], reverse=True)
```

Revenons à notre résolution de problème, c'est à dire trouver les "meilleurs" objets pour la survie.

Idée : La solution optimum est de prendre les 10 valeurs les plus fortes.

Codez ci dessous l'affichage de vos 10 valeurs, les plus fortes.

In [7]:
```
#*complétez ici votre code *
liste_sel = []
for i in range(10):
    liste_sel.append(liste_ordo[i]['objet'])
print(liste_sel)
```

```
['oxygène', 'eau', 'carte', 'aliments', 'emetteur', 'corde', 'seringues', 'parac
hute', 'canot', 'signaux']
```

Mais si l'on rajoute une contrainte de poids ? Les astronautes ne peuvent emporter que 200 kg de matériel.

Nom de l'objet	Valeur	Poids
Une boîte d'allumettes	1	0.1 kg
Des aliments concentrés		2 kg
50 mètres de corde en nylon		5 kg
Un parachute en soie		0.5 kg
Un appareil de chauffage fonctionnant sur l'énergie solaire		25 kg
Deux pistolets calibre 45		0.5 kg
Une caisse de lait en poudre		5 kg
Deux réservoirs de 50 kg d'oxygène chacun	15	100 kg
Une carte céleste des constellations lunaires		0.1 kg
Un canot de sauvetage autogonflable		100 kg
Un compas magnétique		0.5 kg
25 litres d'eau		25 kg
Une trousse médicale et des seringues hypodermiques		0.5 kg
Des signaux lumineux		1 kg
Un émetteur-récepteur fonctionnant sur l'énergie solaire		5 kg

Quelle serait alors la combinaison optimum pour maximiser ses chances de survie ?

Plus difficile, non ?

Formulation mathématique du problème du sac à dos

Toute formulation commence par un énoncé des données. Dans notre cas, nous avons un sac à dos de poids maximal W et n objets. Pour chaque objet i, nous avons un poids w_i et une valeur p_i.

Pour quatre objets ($n = 4$) et un sac à dos d'un poids maximal de 30 kg ($W = 30$), nous avons par exemple les données suivantes :

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

Ensuite, il nous faut définir les variables qui représentent en quelque sorte les actions ou les décisions qui amèneront à trouver une solution. On définit la variable x_i associée à un objet i de la façon suivante : $x_i = 1$ si l'objet i est mis dans le sac, et $x_i = 0$ si l'objet i n'est pas mis dans le sac.

Dans notre exemple, une solution réalisable est de mettre tous les objets dans le sac à dos sauf le premier, nous avons donc : $x_1 = 0$, $x_2 = 1$, $x_3 = 1$, et $x_4 = 1$.

Puis il faut définir les contraintes du problème. Ici, il n'y en a qu'une : la somme des poids de tous les objets dans le sac doit être inférieure ou égale au poids maximal du sac à dos.

Cela s'écrit : $x_1.w_1 + x_2.w_2 + x_3.w_3 + x_4.w_4 \leq W$
ou plus généralement pour n objets :


Pour vérifier que la contrainte est respectée dans notre exemple, il suffit de calculer cette somme : 0 × 13 + 1 × 12 + 1 × 8 + 1 × 10 = 30, ce qui est bien inférieur ou égal à 30, donc la contrainte est respectée. Nous parlons alors de solution réalisable. Mais ce n'est pas nécessairement la meilleure solution.

Enfin, il faut exprimer la fonction qui traduit notre objectif : maximiser la valeur totale des objets dans le sac.

Pour n objets, cela s'écrit :


Dans notre exemple, la valeur totale contenue dans le sac est égale à 10. Cette solution n'est pas la meilleure, car il existe une autre solution de valeur plus grande que 10 : il faut prendre seulement les objets 1 et 2 qui donneront une valeur totale de 11. Il n'existe pas de meilleure solution que cette dernière, nous dirons alors que cette solution est optimale.

Résolution algorithmique - partie 1 - Force brut

Dans un problème d'optimisation chaque solution possède une valeur, et on cherche à déterminer parmi toutes les solutions celle dont la valeur est optimale.

La technique la plus basique pour résoudre ce type de problème consiste à énumérer de façon exhaustive toutes les solutions possibles, puis à choisir la meilleure. Cette approche est nommée par **force brute**.

Vous trouverez ci dessous un extrait du code de résolution du problème de la NASA par force brut. **Attention**, n'executez pas la cellule ci dessous. Le code est incomplet.

Vous trouverez le code complet dans le fichier nasa.py, associé au fichier nasa.csv (qui contient les données.

Le principe de la résolution par force brut est de générer toutes les combinaisons possibles. Puis de sélectionner uniquement celle qui maximise la valeur, tout en respectant les contraintes données.

J'ai utilisé pour résoudre cette problématique par force brute, la bibliothèque itertools pour générer toutes les combinaisons possibles.

```
 allumettes
allumettes\aliments
allumettes\corde
...
allumettes\aliments\corde
allumettes\aliments\parachute
...
```

> Combien y a t'il de combinaisons possible ?

Repondre ici

Au rang 1 : on a 1 éléments dans la liste, il y a 15 combinaisons possibles Au rang 2 : on a deux éléments dans la liste, il y a 105 combinaisons possibles. (attention : carte/eau est la même combinaison que eau/carte) Au rang p : cela fait appel à des notions mathématiques, non vu par vous en première. La formule est

$$C_n^p = \frac{n!}{(n-p)! \times p!}$$

où p est le nombre d'objet dans la combinaison et n le nombre total d'objet. Avec cette formule, on obtient bien 105.

Ici, on cherche la somme de toutes ces combinaisons, et la formule est ici 2^n. Donc 2^{15} = 32 768

> Que pouvez vous en déduire sur la complexité de l'algorithme par force brut ?

La complexite de l'algorithme brut du problème du sac à dos est de l'ordre de 2^n

$2^{poly(n)}$	complexité exponentielle	320 ns	10 µs	10 ms	130 jours	10^{59} ans	...	...	...	problème du sac à dos par force brute

Source : https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes (https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes)

In [ ]:
```
meilleur_valeur=0
meilleur_comb=""
meilleur_poids=0
poids_max = 200

for i in range(len(dicoNasa)):
    for p in itertools.combinations(dicoNasa,i+1) :

        combinaison = listeObjet(p)
        valeurs = sommeValeur(p)
        poids = sommePoids(p)
        #print(combinaison,' ',valeurs,' ',poids,'\n')
        ''''On ne garde que les combinaisons qui respectent les contraintes
        dans notre cas : pas plus de 10 objets pour un poids inférieur à 200
        et en maximisant la valeur
        '''
        if valeurs>meilleur_valeur and poids<=poids_max and len(p)<=10:
            # solution retenue si surpasse la meilleure
            meilleur_valeur = valeurs     # maj meilleure solution
            meilleur_comb = combinaison   # maj meilleure solution
            meilleur_poids = poids        # maj meilleure solution

print("La meilleure sélection est \n",meilleur_comb,"\n valeur = ",meilleur_valeu
r,"\n poids = ",meilleur_poids)
```

La documentation de la bibliothèque Itertools est ici (https://docs.python.org/fr/3/library/itertools.html) https://docs.python.org/fr/3/library/itertools.html (https://docs.python.org/fr/3/library/itertools.html)

Résolution algorithmique - partie 2 - Algo glouton

cours : principes et définitions

Les algorithmes gloutons sont des algorithmes assez simples dans leur logique. Ainsi que leur nom le suggère, ils sont conçus pour prendre le maximum de ce qui est disponible à un moment donné.

L'algorithme glouton choisit la **solution optimale** qui se présente à lui à chaque instant, sans se préoccuper, ni du passé ni de l'avenir. Il répète cette même stratégie à chaque étape jusqu'à avoir entièrement résolu le problème.

Un algorithme glouton est un algorithme qui effectue à chaque instant, le meilleur choix possible sur le moment, sans retour en arrière ni anticipation des étapes suivantes, dans l'objectif d'atteindre au final un résultat optimal.

Les algorithmes gloutons sont parfois appelés algorithmes gourmands ou encore algorithmes voraces. La **répétition** de cette stratégie très simple, permet de résoudre rapidement et de manière souvent satisfaisante des problèmes d'optimisation sans avoir à tester systématiquement toutes les possibilités.

In [ ]:
```
Pseudo Algorithme :
DEBUT
    calculer la valeur (pi / wi) pour chaque objet i,
    trier tous les objets par ordre décroissant de cette valeur,
    sélectionner les objets un à un dans l'ordre du tri et ajouter l'objet sélecti
onné dans le sac si le poids maximal reste respecté.
FIN
```

Vous trouverez ci dessous un extrait du code de résolution du problème de la NASA par la méthode gloutonne. **Attention**, n'executez pas la cellule ci dessous. Le code est incomplet.

Vous trouverez le code complet dans le fichier nasa_glouton.py, associé au fichier nasa.csv (qui contient les données).

In [ ]:
```
# initialisations
cumul_poids = 0
cumul_valeur = 0
choix = []

# on trie la liste mercato par force décroissante  ATTENTION DIFFICULTE ELEVE
# CECI EST L'ETAPE CLE POUR QUALIFIER L'ALGORITHME DE GLOUTON
liste_nasa_ordo =sorted(liste_nasa, key=lambda d: d["valeur"], reverse=True)
print (liste_nasa_ordo)
# on balaye sequentiellement liste_nasa_ordo
for i in range(len(liste_nasa_ordo)):
    # on accumule tant que cela ne va pas dépasser le budget Poids
    # et que l'on ne dépasse pas le nombre d'objet, fixé à 10 par nos contraintes
    if cumul_poids + float(liste_nasa_ordo[i]["poids"]) <= poids_max and len(choi
x)<10:
        cumul_valeur += int(liste_nasa_ordo[i]["valeur"])  #calcul valeur gagnée
        choix.append(liste_nasa_ordo[i])             #complément liste des recrues
        cumul_poids += float(liste_nasa_ordo[i]["poids"])  #calcul poids "dépensée
s"
```

L'exécution du programme est il plus rapide ?

La méthode par glouton est beaucoup plus rapide.

Complexité

Algorithme force brute : On trouve dans le programme une boucle dont la limite dépend de n, on y imbrique une itération .

- pour le pointeur i la gamme (range) est n², parcours de toute la liste
- pour le pointeur p la gamme est n², nombre de combinaisons possibles On peut conclure que le complexité de l'algorithme est O(n3) La complexité en temps croît donc rapidement très supérieur pour l'algorithme force brute

Algorithme glouton : seules les lignes 11 à 17 contiennent des opérations dont le nombre dépend de n (longueur de la liste) si on néglige l'import préalable du fichier .csv On distingue :

- le tri préalable de la liste selon le critère de valeur dont la complexité est O(nlog(n))
- puis le parcours de cette liste triée dont la complexité est O(n) au pire cas

On peut conclure que le complexité de l'algorithme entier est O(n(log(n)+1) qui tend vers une complexité standard O(nlog(n)) si n est grand

Un algorithme glouton ne fournit pas toujours le meilleur résultat possible.

Cas d'usage des algorithmes gloutons

Les algorithmes gloutons sont souvent employés pour résoudre des **problèmes d'optimisation**.

Les algorithmes gloutons se montrent efficaces pour :

- déterminer le plus court chemin dans un réseau
- optimiser la mise en cache de données
- compresser des données
- organiser au mieux le parcours d'un voyageur visitant un ensemble de villes
- organiser au mieux des plannings d'activité ou d'occupations de salles.

Source :

- https://pixees.fr/informatiquelycee/n_site/nsi_prem_glouton_algo.html (https://pixees.fr/informatiquelycee/n_site/nsi_prem_glouton_algo.html)
- https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_sac_%C3%A0_dos (https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_sac_%C3%A0_dos)
- https://www.schoolmouv.fr/cours/algorithmes-gloutons/fiche-de-cours (https://www.schoolmouv.fr/cours/algorithmes-gloutons/fiche-de-cours)
- https://interstices.info/le-probleme-du-sac-a-dos/ (https://interstices.info/le-probleme-du-sac-a-dos/)
- PrépaBac hatier : Algorithme glouton sur le rendu de monnaie
- "Programmation efficace" aux éditions Ellipses
- https://www.lumni.fr/article/jouer-a-survivre-sur-la-lune (https://www.lumni.fr/article/jouer-a-survivre-sur-la-lune)
- http://sdz.tdct.org/sdz/les-algorithmes-gloutons.html (http://sdz.tdct.org/sdz/les-algorithmes-gloutons.html)
- https://pixees.fr/informatiquelycee/n_site/nsi_prem_glouton_algo.html (https://pixees.fr/informatiquelycee/n_site/nsi_prem_glouton_algo.html)

In [ ]: