Algorithmes gloutons

1. Présentation

Il y a essentiellement trois grands paradigmes pour concevoir un algorithme :

- Les stratégies gloutonnes (ce qui sera vu dans cette feuille).
- Diviser pour régner (par exemple la recherche dichotomique).
- La programmation dynamique (c'est au programme de terminale).

Par paradigme, on entend une façon de concevoir un algorithme pour résoudre un problème.

Les stratégies gloutonnes sont très naturelles dans leur approche et permettent de résoudre une grande variété de problèmes.Les algorithmes gloutons sont abordés ici à l'aide de problèmes classiques :

- Le problème du rendu de monnaie.
- Le problème du sac à dos.
- La plannification d'activités.

Définition:

Un algorithme glouton détermine une solution après avoir effectué une série de choix. Pour chaque point de décision, il retient le choix qui semble le meilleur à cet instant. Il ne revient ensuite pas sur ce choix. Cette stratégie ne produit pas toujours une solution optimale.

Exercice 1 :

On considère une liste d'objets dont voici les masses, en kg : 7 , 6 , 3 , 4 , 8 , 5 , 9 , 2. On place ces objets dans des cartons de 11 kg maximum.

1. En remplaçant les tirets par les masses des objets, déterminer le nombre minimum de cartons que l'on peut faire.

Carton 1	Carton 2	Carton 3	Carton 4	Carton 5	Carton 6
-	-	-	-	-	-
-	-	-	-	-	-

1. Décrire une stratégie gloutonne qui permet d'aboutir au résultat.

Réponses :

1.
2.

Exercice 2 :

Même objectif avec les masses suivantes : 6 , 5 , 5 , 3 , 3 , 2. On cherche cette fois à faire des cartons de 12kg.

1. Compléter ce tableau en utilisant la même stratégie gloutonne que précédemment.

Carton 1	Carton 2	Carton 3	Carton 4
-	-	-	-
-	-	-	-
-	-	-	-

1. Montrer que cette solution n'est pas optimale.

Réponses :

1.
2.

A retenir :

- Les algorithmes gloutons sont intuitifs, simples à concevoir et peu coûteux en temps et mémoire.
- Ils répondent à de nombreux problèmes d'optimisation(plus court chemin, planification de tâches...)
- Ils produisent une solution à des problèmes difficiles, même si celle-ci n'est pas toujours optimale.

2. Problème du rendu de monnaie

Le problème du rendu de monnaie est un problème classique d'algorithmique. Il s'énonce de la façon suivante : étant donné un système de monnaie (pièces et billets), comment rendre une somme donnée de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets ?

Par exemple, la meilleure façon de rendre 7 euros est de rendre un billet de cinq et une pièce de deux, même si d'autres façons existent (rendre 7 pièces de un euro, par exemple).

Exercice 3 :

Le système monétaire européen est constitué des valeurs suivantes (pour simplifier, on ne tient pas compte du billet de 500€) :

`[200,100,50,20,10,5,2,1]` . On suppose que l'on dispose d'autant de pièces que l'on veut.

1. Compléter le tableau ci-dessous en indiquant pour chaque somme S, le nombre de pièces minimal et les pièces utilisées.

Somme S	9	46	149
Nombre minimal de pièces	3	-	-
Pièces utilisées	[5,2,2]	-	-

1. Quelle stratégie gloutonne emploie-t-on intuitivement ?
2. Déterminer une somme pour laquelle cette stratégie ne renvoie pas une solution optimale.

Réponses :

1.
2.

A retenir :

- La méthode « usuelle » pour rendre la monnaie est celle de l'algorithme glouton : tant qu'il reste quelque chose à rendre, choisir la plus grosse pièce qu'on peut rendre (sans rendre trop). C'est un algorithme très simple et rapide, et on appelle canonique un système de pièces pour lequel cet algorithme donne une solution optimale quelle que soit la valeur à rendre.
- Il se trouve que presque tous les systèmes de pièces réels de par le monde sont canoniques (dont celui de l'euro).

Programmation

Exercice 4 :

Compléter la fonction `rendu(S,syst)` qui prend en paramètre la somme S à rendre, de type entier et un système de pièces sous forme d'une liste d'entiers. Cette fonction doit renvoyer un tuple constitué de nombre de pièces minimal et de la liste contenant les pièces rendues. Ainsi, si $S = 9$ et si `syst` contient le système européen, l'appel de `print(S,syst)` doit renvoyer `(3, [5, 2, 2])` .

In [30]:

```
def rendu(S,syst):
    i=0
    rendu=[]




    return (len(rendu), rendu)

#système européen
syst=[200,100,50,20,10,5,2,1]
#somme à rendre
S=9

print(rendu(S,syst))
```

```
(0, [])
```

Exercice 5 :

La livre sterling est l'une des plus anciennes monnaies qui a encore cours avec une histoire de près de 1 000 ans. Jusqu'en 1971, la livre était divisée en 20 shillings et un shilling valait 12 pence.Encore plus compliqué, il y avait les sous-unités de la livre : le shilling (1/20 de livre), le florin (1/10 de livre ou 2 shillings), la demi-couronne (1/8 de livre ou 2 shillings et demi), la couronne (1/4 de livre ou 5 shillings), le demi-souverain (1/2 livre) et le souverain or qui valait une livre.

Si l'on considère que le penny est la plus petite unité, cela revient à raisonner avec le système de pièces suivant : `[240,120,60,30,24,12,1]`

Eventuellement à l'aide de la fonction précédente, trouver un contre exemple qui permet de de démontrer que l'algorithme glouton ne renvoie pas nécéssairement une solution optimale.

In [21]:

```
#Système anglais avant 1971
syst=[240,120,60,30,24,12,1]
S=0

print(rendu(S,syst))
```

```
(0, [])
```

Réponse :

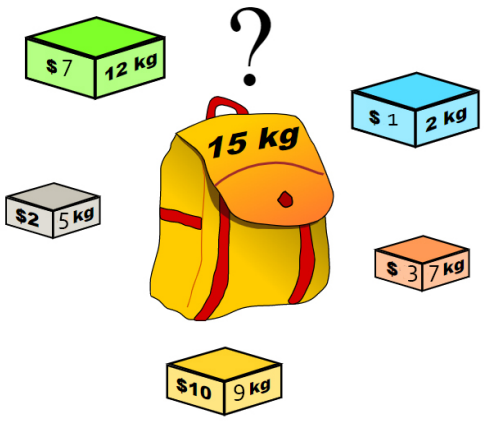
- Somme S :
- Liste des pièces renvoyées par l'algorithme :
- Solution optimale :

3. Problème du sac à dos

Le problème du sac à dos, noté également KP (en anglais, Knapsack problem) est un problème classique d'optimisation. Il modélise une situation analogue au remplissage d'un sac à dos, ne pouvant supporter plus d'un certain poids, avec tout ou partie d'un ensemble donné d'objets ayant chacun un poids et une valeur. Les objets mis dans le sac à dos doivent maximiser la valeur totale, sans dépasser le poids maximum.

Exercice 6:

Un cambrioleur en pleine action souhaite remplir son sac d'une capacité maximale de 15kg avec des objets qui se trouvent devant lui:



- Une drôle de statue en cuivre d'une valeur de 10 dollars, qui pèse 9 kg.
- Une enclume de voyage qui pèse 12 kg, d'une valeur de 7 dollars.
- Un grille pain qui pèse 2 kg, d'une valeur de 1 dollars.
- Une console de jeux de 1985 soldée à 3 dollars et qui pèse 7 kg.
- L'encyclopédie des nains de jardin qui pèse 5 kg et vendue 2 dollars.

1. Quelle(s) stratégies(s) gloutonnes peut-il mettre en oeuvre ?
2. Finalement à l'aide d'une de ces stratégies, quels objets va-t-il prendre pour maximiser la valeur transportée ?

Réponse :

1. On peut envisager 3 stratégies :
    -
    -
    -

1.
    -

Programmation

- On représente un objet par une liste trois élements (nom de l'objet, poids , valeur). Ainsi l'enclume est représentée par `['Enclume', 12,7]` .
- La liste des objets est donc une liste de listes :
  - `[['Statue',9,10],['Enclume',12,7],['Grille pain',2,1],['Console',7,3], ['Encyclopédie',5,2]]`
- `P` est le poids maximal (ici 15).

Ne pas oublier d'éxécuter cette cellule

In [13]:

```
objets=[['Statue',9,10],['Enclume',12,7],['Grille pain',2,1],
        ['Console',7,3], ['Encyclopédie',5,2]]
P=15
```

Exercice 7 :

Exécuter la cellule ci-dessous et décrire ce que fait la fonction `valeurs_kg(objets )`.

In [14]:

```
def valeurs_kg(objets):
    objets=[[objet[2]/objet[1]]+objet for objet in objets]
    objets.sort(reverse=True)

    return objets

print(valeurs_kg(objets))
```

```
[[1.1111111111111112, 'Statue', 9, 10], [0.5833333333333334, 'Enclume', 1
2, 7], [0.5, 'Grille pain', 2, 1], [0.42857142857142855, 'Console', 7, 3],
[0.4, 'Encyclopédie', 5, 2]]
```

Réponse :

-
-
-

Exercice 8:

Compléter la fonction `sac_glouton(P,objets)` qui prend en paramètres le poids P maximal autorisé et la liste des objets possibles.Cette fonction doit renvoyer le contenu du sac du cambrioleur obtenu avec la stratégie gloutonne utilisée dans l'exercice 6.

Ainsi `sac_glouton(P,objets)` doit renvoyer `[[1.1111111111111112, 'Statue', 9, 10], [0.5, 'Grille pain', 2, 1]]` , ce qui correspond bien à la réponse trouvée précédemment.

In [16]:

```
def sac_glouton(P,objets):
    objets=valeurs_kg(objets)
    sac=[]
    poids_sac=0




    return sac

print(sac_glouton(P,objets))
```

```
[]
```

Exercice 9 :

La solution renvoyée par l'algorithme glouton dans le problème du sac à dos n'est pas optimale. Trouver un contre exemple.

Réponse :

-

A retenir :

- Hors de ce contexte amusant, on utilise aussi ce problème pour modéliser de nombreuses situations, quelquefois en tant que sous-problème :
  - Dans les systèmes financiers, où l'idée est la suivante : étant donné un certain montant d'investissement dans des projets, quels projets choisir pour que le tout rapporte le plus d'argent possible.
  - Pour la découpe de matériaux, afin de minimiser les pertes dues aux chutes.
  - Dans le chargement de cargaisons (avions, camions, bateaux…).
  - Ou encore, dès qu'il s'agit de préparer une valise ou un sac à dos pour une randonnée...
- Il existe deux grandes catégories de méthodes de résolution de problèmes d'optimisation de ce type : les méthodes approchées et les méthodes exactes.
  - Les méthodes approchées, encore appelées heuristiques, permettent d'obtenir rapidement une solution approchée(comme ici avec un algorithme glouton), mais pas nécessairement optimale.
  - Les méthodes exactes permettent d'obtenir la solution optimale à chaque fois, mais le temps de calcul peut être long si le problème est compliqué à résoudre ou s'il comporte un grand nombre de données.

Exercice 10 (Pour aller plus loin) : Programmation d'une méthode exacte, la recherche par force brute.

La recherche par force brute consiste à tester toutes les possibilités pour identifier la meilleure. Reprenons nos objets et notre sac :

In [ ]:

```
objets=[['Statue',9,10],['Enclume',12,7],['Grille pain',2,1],['Console',7,3], ['Encyclo
pédie',5,2]]
P=15
```

1. Sans tenir compte de la contrainte du poids ou de la valeur, combien de sacs différents peut-on faire avec 5 objets ? Justifier.

Réponse :

-

1. Compléter la fonction `forcebrute(objets)` qui prend en paramètre une liste d'objets et qui renvoie la liste de tous les sacs possibles sans tenir compte de la contrainte du poids ou de la valeur.

Remarques :

- On pourra considérer d'abord le premier objet, constituer deux sacs (un avec cet objet et l'autre vide), puis considérer le deuxième objet qui sera ajouté ou pas aux deux sacs précédents pour constituer deux nouveaux sacs, etc...
- Lorsqu'un objet n'est pas ajouté dans un sac, on pourra utiliser la liste `['Rien', 0, 0]` pour décrire son absence dans le sac. Ainsi `[['Statue',9,10],['Rien',0,0],['Grille pain',2,1], ['Rien',0,0], ['Encyclopédie',5,2]]` contient 3 objets.

In [1]:

```
#2.bruteforce
def forcebrute(objets):
    sacs=[[]]
    for i in range(len(objets)):
        sacs_next=[]




        sacs=sacs_next

    return sacs
```

1. Compléter les fonctions :
   - `poids(sac)` qui prend en paramètre une liste d'objets et qui renvoie le poids du sac ainsi constitué.
   - `valeur(sac)` qui prend en paramètre une liste d'objets et qui renvoie la valeur du sac ainsi constitué.

In [2]:

```
#3.
def poids(sac):
    total=0



    return total

def valeur(sac):
    total=0



    return total
```

1. Compléter la fonction `meilleur_sac(P, objets)` qui prend en paramètres une liste d'objets et un poids maximal P, et qui renvoie le contenu du sac ayant la plus grande valeur tout en respectant la contrainte de poids.

In [ ]:

```
def meilleur_sac(P,objets):
    sacs=forcebrute(objets)
    top_sac=sacs[0]




    return top_sac



solution=meilleur_sac(P,objets)
print(solution)
```

4. Choix d'activités

Le problème du choix d'activités est également un grand classique des problèmes d'optimisation. Il peut s'énoncer ainsi : Etant donné un ensemble d'activités, chacune possédant une date de début et une date de fin, comment choisir le maximum d'activités compatibles entre elles ?

Exercice 11 :

Le festival Alphabet se déroule tous les ans. Ce festival commence tôt le matin,dès 5h, et se termine à minuit. Des artistes hors normes sont à l'affiche cette année :

- Alpha se produira de 8h à 10h. Après une courte pause, il donnera un deuxième spectacle de 13h à 17h. Puis, il reviendra plus tard dans la soirée de 21h à 24h.
- La grande artiste Beta donnera également trois représentations. La première de 6h à 9h, la deuxième de 10h à 14h et la dernière de 17h à 18h.
- Gamma viendra nous honorer de leur présence. Ils donneront deux spectacles : l'un de 11h à 15h, et l'autre de 18h à 21h.
- Le célèbre Delta donnera deux concerts : le premier de 5h à 11h et le second de 13h à 16h.
- Le groupe de musique local Epsilon se produira de 8h à 13h puis de 20h à 22h.
- La troupe de théâtre Zeta assurera un spectacle continu entre 7h et 18h.
- Enfin, le groupe Eta assurera le show de 10h à 12h.

	Alpha	Bêta	Gamma	Delta	Epsilon	Zêta	Êta
5h-6h							
6h-7h							
7h-8h							
8h-9h							
9h-10h							
10h-11h							
11h-12h							
12h-13h							
13h-14h							
14h-15h							
15h-16h							
16h-17h							
17h-18h							
18h-19h							
19h-20h							
20h-21h							
21h-22h							
22h-23h							
23h-24h							

Alcide souhaite assister au maximum de spectacles possibles, quitte à voir plusieurs fois le même artiste. Il arrive dès le dèbut du festival et il sait qu'il n'est pas possible de sortir avant la fin d'un spectacle.

1. Donner trois stratégies gloutonnes qui permettent d'aboutir à une sélection de spectacles.
2. Donner, pour chaque stratégie, la sélection de spectacles correspondante.
3. Quelle sélection conseillller à Alcide ?

Réponses.

1. Stratégies :
   - A :
   - B :
   - C :

1. Sélections :
   - A :
   - B :
   - C :

1. La stratégie B donne un meilleur résultat(6 spectacles) :
   -

Exercice 12: Programmation de l'algorithme

Partie A : Choix de la structure de données.

On modélise chaque spectacle par un triplet dont le premier élément est le nom de l'artiste(de type 'string', le second l'heure de début(de type entier) et le troisième l'heure de fin(de type entier). Le programme du festival sera modélisé par la liste `Alphabet` .

1. Compléter ci-desous cette liste :

In [1]:

```
#Programme du festival
Alphabet=[('Alpha',8,10)]
```

1. Afficher l'heure de fin du deuxième spectacle d'Alpha.

In [3]:

```
#2.
```

```
17
```

Partie B : Compatibilité des spectacles

On dit que deux spectacles sont compatibles si l'horaire de fin du premier spectacle est inférieure ou égale à l'horaire de début du second spectacle.

Compléter la fonction `compatibles(spect1, spect2)` qui prend en paramètres deux spectacles et qui renvoie `True` si deux spectacles sont compatibles, `False` sinon.

In [4]:

```
def compatibles(spect1, spect2):
    return "A compléter"

#tests
spect1=('Alpha',8,10)
spect2=('Beta',10,14)
spect3=('Epsilon',8,13)

#true
print(compatibles(spect1,spect2))
print(compatibles(spect2,spect1))

#false
print(compatibles(spect1,spect3))
print(compatibles(spect3,spect1))
```

```
A compléter
A compléter
A compléter
A compléter
```

Partie C : Algorithme glouton

1. La fonction ci-dessous renvoie une liste de tuples triée, ordre croissant à l'aide du dernier élément de chaque tuple.A l'aide de cette foncion afficher les spectacles du festival triès par horaire de fin.

In [6]:

```
def tri_heure_fin(festival):
    return sorted(festival,key=lambda spect:spect[2])
```

1. Compléter la fonction `prochain_spect(spect, festival)` qui prend en paramètres un spectacle et une liste de spectacles triées par horaires de fin. Cette fonction renvoie le prochain spectacle compatible avec l'horaire de fin de plus proche.S'il n'y en a pas, la fonction renvoie `None` .

Ainsi:

- `prochain_spect(('Beta',10,14),festival)` doit renvoyer `('Beta', 17, 18)`
- `prochain_spect(('Epsilon',20,22),festival)` doit renvoyer `None`

In [8]:

```
def prochain_spect(spect, festival):
    i=festival.index(spect)  #index de l'élément spect dans festival




    return "A compléter"

#tests
spect1=('Beta',10,14)
spect2=('Epsilon',20,22)
festival=tri_heure_fin(Alphabet)

print(prochain_spect(spect1,festival))
print(prochain_spect(spect2,festival))
```

```
A compléter
A compléter
```

1. Compléter la fonction `selection(festival)` qui prend en paramètre une liste de spectalces et qui renvoie la liste des spectacles compatibles à l'aide de la stratégie gloutonne selon l'horaire de fin.

L'appel de `selection(Alphabet)` doit renvoyer :

- `[('Beta', 6, 9), ('Eta', 10, 12), ('Delta', 13, 16), ('Beta', 17, 18), ('Gamma', 18, 21), ('Alpha', 21, 24)]` .

In [10]:

```
def selection(festival):
    festival=tri_heure_fin(festival)
    prochain=festival[0]
    select=[prochain]









    return select

print(selection(Alphabet))
```

```
[('Beta', 6, 9)]
```

In [ ]:

```
def selection(festival):
    festival=tri_heure_fin(festival)
    prochain=festival[0]
    select=[prochain]

    for i in range(len(festival)):
        prochain=prochain_spect(prochain,festival)
        if prochain is not None:
            select.append(prochain)
            i=festival.index(prochain)
        else:
            return select


    return select

print(selection(Alphabet))
```

FIN