

Objectifs

- Découvrir la notion d'algorithme « glouton ».
- Etudier au travers de divers exemples, si un tel algorithme fournit une solution optimale à un problème proposé.

Exercice 1 Rendu de monnaie (20mn) [Sol 1] Le problème du rendu de monnaie est un problème d'algorithmique qui s'énonce de la façon suivante : étant donné un système de monnaie, comment rendre une somme donnée de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets ?

Dans la zone euro, le système en vigueur, en mettant de côté les centimes d'euros, met à disposition des pièces ou billets de : 1 €, 2 €, 5 €, 10 €, 20 €, 50 €, 100 €, 200 €, 500 €.

On suppose que nous avons à notre disposition un nombre illimité de ces pièces ou billets.

1) Premier contact :

- 1.1)** Supposons que la somme à rendre est de 7€. Faire la liste de toutes les façons de rendre une telle somme. Quel est le nombre minimal de pièces ou billets à rendre ?
 - 1.2)** Supposons que la somme à rendre soit maintenant de 463 €. Trouver une méthode permettant d'effectuer un rendu de monnaie en utilisant le moins possible de pièces sans avoir à énumérer toutes les possibilités. Combien de pièces ou de billets semblent nécessaires ?
Il est très probable qu'en résolvant cette dernière question, vous ayez mis en oeuvre la méthode dite « gloutonne » : tant qu'il reste quelque chose à rendre, on choisit la plus grosse pièce (ou plus gros billet) qu'on peut rendre (sans rendre trop).
- 2) Est-ce toujours une bonne stratégie?** Supposons que le système de monnaie mette à disposition uniquement des pièces de 1 €, 3 € et 4€ (on ne tient tou-

jours pas compte des centimes d'euros) et supposons que nous devons rendre la somme de 6 €. Combien de pièces l'algorithme glouton nous amènerait-il à rendre ? Qu'en pensez-vous ? Pour le système monétaire européen (qui ne comporte aucune pièce ni aucun billet de 3€ ni de 4€), une telle situation ne se produirait pas ! Pour cette raison, un tel système de monnaie est qualifié de canonique.

Sans s'en rendre compte, tout individu met en oeuvre un algorithme glouton pour rendre la monnaie.

3) Mise en oeuvre d'une solution avec Python Un exemple incomplet de mise en oeuvre de l'algorithme avec python est proposé ci-dessous :

```
systeme_monnaie = [500,200,100,50,20,10,5,2,1]
# valeurs des pièces dans l'ordre décroissant
somme_a_rendre = 99 # somme à rendre
liste_pieces = [] # liste des pièces à rendre
i = 0 # indice de la première pièce à rendre
while somme_a_rendre > 0: # Construction de la liste des \
↳ pièces
    valeur = systeme_monnaie[i]
    if somme_a_rendre < valeur:
        i += 1
    else:
        ...
        # ajouter la pièce à la liste des pièces à rendre
        ...
        # Mettre à jour la nouvelle somme à rendre
return(liste_pieces)
```

3.1) Compléter les deux lignes en pointillé.

3.2) Écrire une fonction qui reçoit deux arguments (la somme à rendre et le système de monnaie) et qui renvoie la liste des pièces choisies par l'algorithme glouton. Testez cette fonction sur quelques exemples.

**À retenir**

L'algorithme glouton est un algorithme qui suit le principe de faire à chaque étape, un choix optimum local. L'algorithme agit en quelque sorte de manière « gloutonne » : devant un nouveau choix à faire, il choisit l'option qui apparaît être la meilleure à ce moment là. Cependant, comme cela a été vu dans la question 2, l'utilisation de cette méthode n'est pas toujours optimale.

Exercice 2 Le problème du sac à dos (20mn) [Sol 2] Le « problème du sac à dos » est un autre exemple classique de problème d'optimisation. Il peut s'énoncer ainsi : on dispose devant soi, d'objets ayant chacun une masse et une valeur, et d'un sac ne pouvant supporter plus d'une certaine masse. Il s'agit de remplir le sac en maximisant la valeur totale des objets et sans dépasser la masse maximale.

1) Premier contact : Supposons que l'on dispose d'un sac à dos de contenance maximum 30 kg et de quatre objets A, B, C et D dont les caractéristiques sont les suivantes :

Objet	A	B	C	D
Masse	13 kg	12 kg	8 kg	10 kg
Valeur	70 €	40 €	30 €	30 €

1.1) Combien doit-on tester de combinaisons dans ce cas là pour remplir au mieux le sac à dos?

1.2) Quelle semble être la solution optimale?

2) Construction d'une solution réalisable : Lorsque le nombre d'objets devient important, la solution précédente consistant à tester toutes les combinaisons devient inutilisable. En effet, s'il y a N objets, on peut démontrer qu'il y a $2^N - 1$ combinaisons à tester et, même pour un ordinateur très puissant, le temps de calcul devient trop important. On peut alors se tourner vers des méthodes qui fournissent rapidement une solution réalisable mais pas nécessairement optimale. C'est le cas de la méthode gloutonne. Pour cela, on introduit la notion « d'efficacité d'un objet » comme étant le rapport de sa valeur sur sa masse. Plus la valeur de l'objet est importante par rapport à sa masse, plus l'objet sera dit « efficace ».

2.1) Calculer l'efficacité de chacun des objets A, B, C et D évoqués précédemment, puis classer ces objets par ordre d'efficacité.

2.2) Proposer alors une méthode gloutonne permettant de résoudre le problème.

3) Mise en oeuvre d'une solution avec Python : On suppose créée une variable Max qui contient la capacité maximale (en kilogrammes) du sac à dos. On suppose également construites 3 listes liste_noms, liste_masses et liste_valeurs qui contiennent respectivement les noms, masses et valeurs de chacun des objets. Par exemple, avec les objets utilisés précédemment :

```
# Liste des noms des objets
liste_noms = ["A", "B", "C", "D"]
# Liste des masses des objets
liste_masses = [13, 12, 8, 10]
# Liste des valeurs des objets
liste_valeurs = [70, 40, 30, 30]
```

3.1) Créer une liste L dont le *i*-ème élément est une liste contenant respectivement l'efficacité du *i*-ème élément, son nom, sa masse et sa valeur.

3.2) Trier la liste L selon les efficacités par ordre décroissant (en attendant de voir très prochainement les algorithmes de tris, on pourra utiliser la commande `L_triee=sorted(L, reverse=True)`).

3.3) Compléter alors les lignes en pointillé dans le script ci-dessous

```
liste_objet=[]
# Initialisation de la liste des objets rangés dans le sac
i = 0
# indice du premier objet
Somme=0
# somme des masses des objets déjà rangés
for i in range(len(L_triee)):
    if Max >= ...:
        liste_objet.append(...)
        Somme = Somme+...
```

La méthode gloutonne ne fournit pas dans notre exemple une solution optimale. En effet, nous avons vu plus haut que les objets A et B permettent d'atteindre une valeur de 110 € pour une masse de 25 kg, alors que la méthode gloutonne propose de prendre les objets A et C pour un montant total de 100 € et une masse totale de 21 kg.

**À retenir**

A nouveau insistons sur le fait qu'en général, un algorithme glouton ne donne pas une solution optimale à un problème d'optimisation. Cependant, l'algorithme glouton fournit rapidement une solution réalisable.

Exercice 3 La fête du cinéma (50mn) [Sol 3] Le jour de la fête du cinéma, vous voulez aller voir un maximum de film dans la journée. Chaque film est caractérisé par un intervalle $[d, f[$ où d et f sont respectivement les heures de début et de fin. Deux films F_1 et F_2 sont alors compatibles si leur intervalle $[d_1; f_1[$ et $[d_2; f_2[$ sont disjoints. Quatre cinémas proposent des films :

- Le cinéma 1 propose 4 films :

$F_1 : [3, 4[$ $F_2 : [0, 1[$ $F_3 : [2, 3[$ $F_4 : [1, 2[$

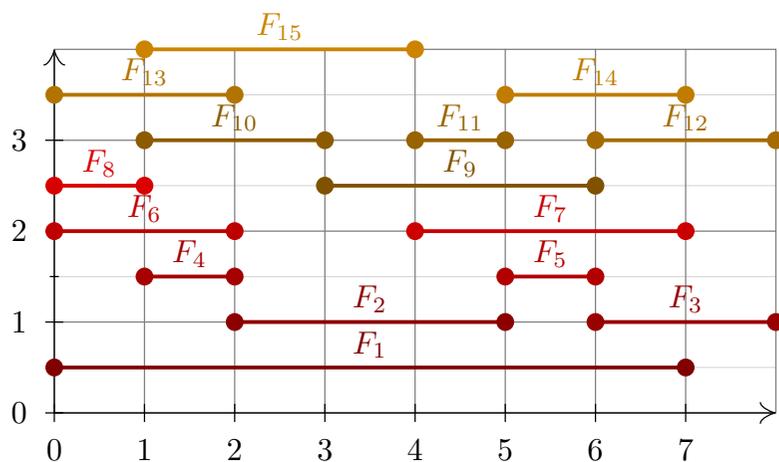
- Le cinéma 2 propose également 4 films :

$F_1 : [2, 4[$ $F_2 : [0, 1[$ $F_3 : [1, 3[$ $F_4 : [0, 2[$

- Le cinéma 3 propose 3 films :

$F_1 : [0, 3[$ $F_2 : [1, 2[$ $F_3 : [2, 3[$

- Le cinéma 4 propose une belle affiche composée de 15 films, représentée par le diagramme ci-dessous :



Nous allons envisager trois algorithmes gloutons différents, suivant nos priorités.

1) Un premier algorithme glouton : Dans cette question, la priorité de l'algorithme glouton est d'aller voir le film qui commence en premier puis, une fois le film terminé, d'attendre le moins possible avant d'aller voir un autre film (dans le cas de films qui commencent à la même heure, c'est le film F_i avec le i minimal qui est choisi). Tester à la main cet algorithme glouton pour chacun des 4 cinémas.

2) Un deuxième algorithme glouton : Dans cette question, la priorité de l'algorithme glouton est d'aller voir le film le plus court, puis une fois le film terminé, aller voir le film le plus court restant... (Là encore, dans le cas de films qui commencent à la même heure, c'est le film F_i avec le i minimal qui est choisi). Tester à la main cet algorithme glouton pour chacun des 4 cinémas.

3) Un troisième algorithme glouton : Dans cette question, la priorité de l'algorithme glouton est d'aller voir le film qui se termine en premier.

3.1) Créer un cinéma aléatoire, c'est-à-dire, une fonction

```
Programmation(debut:int, fin:int, nb:int) ->list
```

à qui on fournit une heure *debut* de début de séance, une heure *fin* de fin de séance et un nombre entier *nb* de films, et qui renvoie une liste aléatoire de *nb* éléments de la forme $["F_j", d_j, f_j]$ où F_j désigne le nom du film, d_j son horaire de départ et f_j son horaire de fin, tous les films devant commencer et se terminer entre *debut* (inclus) et *fin* (inclus).

3.2) (*) On suppose créée une liste *Prog* du type précédent. L'algorithme suivant permet de trier les films par horaire de fin croissant. Il s'appuie sur la méthode de tri dite "par insertion". S'il vous reste du temps, commentez chacune des lignes du script :

```
n=len(Prog)
for i in range(1,n):
    j = i
    Prog_i = Prog[i]
    elt = Prog[i][2]
    while (j > 0) and (elt < Prog[j-1][2]):
        Prog[j] = Prog[j-1]
        j = j-1
    Prog[j] = Prog_i
```

3.3) Écrire une fonction `Gloutons3(Prog:list) ->list` à qui on fournit une liste (telle que décrite dans les questions précédentes) et qui renvoie une liste de films obtenus par la méthode gloutonne proposée. Tester cette fonction sur plusieurs exemples générés par la fonction `Programmation`



À retenir

En résumé :

- Un algorithme glouton est un algorithme utilisé pour résoudre un problème d'optimisation où l'on fait à chaque étape le « meilleur choix local » (charge au concepteur de l'algorithme de définir ce qu'est un meilleur choix local)
- Les algorithmes gloutons ne sont pas toujours adaptés : pour certains problèmes, ils ne permettent pas de donner une solution optimale.
- Lorsqu'un algorithme glouton permet de trouver une solution optimale à un problème, cet algorithme est souvent un excellent choix.

Solution 1

1) 1.1) On peut proposer les façons suivantes :

1+1+1+1+1+1+1

1+1+1+1+1+2

1+1+1+2+2

1+2+2+2

1+1+5

2+5

Parmi toutes ces possibilités, la dernière est celle qui nécessite le moins de pièces ou de billets (seulement 2).

1.2) On peut commencer par utiliser un billet de 200€. Il reste alors à rendre 263€.

On peut alors utiliser un autre billet de 200€. Il reste alors 63€ à rendre.

On choisit alors un billet de 50€. Il reste alors 13€ à rendre.

On utilise alors un billet de 10€. Il reste alors 3€ à rendre.

On utilise alors une pièce de 2€ puis une pièce de 1€.

Au total, avec ce choix, 6 pièces ou billets ont été nécessaires.

2) L'algorithme décrit dans la question précédente nous conduirait à utiliser d'abord une pièce de 4€ puis 2 pièces de 1€. Ce n'est pas une solution optimale puisque cette solution demande d'utiliser 3 pièces alors qu'on peut rendre la monnaie avec seulement 2 pièces (3€ + 3€).

3) 3.1) Les deux pointillés pourraient être respectivement remplacés par

```
liste_pieces.append(valeur)
```

et

```
somme_a_rendre=somme_a_rendre-valeur.
```

3.2) On peut proposer la solution suivante :

```
def rendu_monnaie(somme:int,systeme:list)->list:
    liste_pieces = [] #Liste des pièces à rendre
    somme_a_rendre=somme
    i=0 #indice de la première pièce à rendre
    while somme_a_rendre > 0: #Construction de la liste des
    ↪ pièces
        valeur = systeme[i]
        if somme_a_rendre < valeur:
            i += 1
        else:
            liste_pieces.append(valeur)
            somme_a_rendre=somme_a_rendre-valeur
    return(liste_pieces)
```

Solution 2

1) 1.1) On peut tester :

- Les 4 objets en même temps : un seul test
- 3 objets (A,B et C; A,B et D; A,C et D; B, C et D) : 4 tests
- 2 objets : 6 tests
- 1 seul objet : 4 tests

Cela fait donc au total 15 tests à effectuer.

1.2) La solution optimale semble être de prendre les objets A et B : la somme de leur masse fait moins de 30 kg et la valeur totale de ces objets est de 110€.

2) 2.1) Les efficacités des objets A, B, C et D sont respectivement égales à $\frac{70}{13} \approx 5,38$,

$\frac{40}{12} \approx 3,33$, $\frac{30}{8} = 3,75$ et $\frac{30}{10} = 3$, ce qui donne :

efficacité(D) < efficacité(B) < efficacité(C) < efficacité(A)

2.2) On pourrait proposer la méthode gloutonne suivante : tant qu'il reste de la place dans le sac à dos, on choisit l'objet ayant la plus grande efficacité (et dont la masse ne fait pas dépasser la capacité maximale du sac).

3) 3.1) On peut proposer le script suivant :

```
L=[]
for i in range(len(liste_noms)):
    w = liste_valeurs[i]/liste_masses[i]
    x = liste_noms[i]
```

```
y = liste_masses[i]
z = liste_valeurs[i]
L.append([w, x, y, z])
```

3.2) Il n'y a qu'à écrire : `L_triee = sorted(L, reverse=True)`.

3.3) On peut proposer le script suivant :

```
liste_objet=[]
# Initialisation de la liste des objets rangés dans le sac
i = 0
# indice du premier objet
Somme=0
# somme des masses des objets déjà rangés
for i in range(len(L_triee)):
    if Max >= Somme+L_triee[i][2] :
        liste_objet.append(L_triee[i][1])
        Somme = Somme + L_triee[i][2]
```

Solution 3

- 1) ● Pour le cinéma 1, le film qui commence en premier est F2, puis une fois sorti, on peut aller voir F4, puis F3 puis F1.
 - Pour le cinéma 2, F2 et F4 commencent en premier, donc, d'après la règle proposée, on va voir F2. Le film qui commence alors en premier est F3.
 - Pour le cinéma 3, on va voir seulement F1.
 - Pour le cinéma 4, on va voir seulement F1.
- 2) ● Pour le cinéma 1, les 4 films ont la même durée. D'après la règle proposée, on va donc voir F1 puis la séance est finie!
 - Pour le cinéma 2, on va d'abord voir F2 puis F1.
 - Pour le cinéma 3, on va d'abord voir F2 puis F3.
 - Pour le cinéma 4, on commence par F4 puis F5 puis F3.

3) 3.1) On peut proposer la fonction suivante (après avoir importé le module numpy) :

```
def Programmation(debut:int,fin:int,nb:int)->list:
    L = []
    for j in range(nb):
        dj = np.random.randint(debut,fin)
        fj = np.random.randint(dj+1,fin+1)
        Fj = "F"+str(j+1)
        L.append([Fj, dj, fj])
    return(L)
```

3.2) L'idée du tri par insertion est la suivante : On parcourt le tableau à trier du début à la fin (ligne 2). Au moment où on considère le *i*-ème élément, tous les éléments qui le précèdent sont déjà triés. On insère alors ce *i*-ème élément à sa place parmi ceux qui le précèdent. Pour cette dernière étape, on garde la valeur du *i*-ème élément (ligne 4) ainsi que de l'heure de fin de ce *i*-ème élément (ligne 5), et on recherche parmi les éléments précédents où ce *i*-ème élément va devoir être inséré (lignes 6 à 8).

3.3) La fonction suivante convient :

```
def Gloutons3(Prog:list)->list:
    n = len(Prog) #Nombre de films au programme

    for i in range(1, n): #Tri par ordre de fin croissante
        j = i
        elt = Prog[i][2]
        Progi = Prog[i]
        while (j > 0) and (elt < Prog[j-1][2]):
            Prog[j] = Prog[j-1]
            j = j-1
        Prog[j] = Progi

    Planning = [Prog[0]] #Planning de la journée
    j = 0
    for i in range(1, n):
        if Prog[i][1] >= Prog[j][2]: #Si le film commence \
            ↪ après la fin du film j
            Planning.append(Prog[i])
            j = i
    return(Planning)
```