Travaux pratiques 7

Algorithmes Gloutons

Informatique tronc commun MPSI et PCSI

Faire toujours un choix localement optimal dans l'espoir que ce choix mènera à la solution optimale du problème. *Devise de l'algorithme glouton*

Recopier le fichier `TP7.py` présent dans le répertoire public de votre classe. Ce fichier contient toutes les variables et importations nécessaires à ce TP.

I Problème 1 : le voyageur de commerce

I.1 Présentation

Un commercial a plusieurs rendez vous et doit se rendre dans plusieurs villes en voiture.
Il cherche l'itinéraire (l'ordre des villes) minimisant la distance totale parcourue.
Il s'autorise à visiter les villes dans n'importe quel ordre, mais aucune ne doit être négligée.
Il faudra à la fin revenir à la ville de départ.
Il part de Nancy et a des rendez vous dans les villes suivantes : Metz, Paris, Reims et Troyes.

Villes	Nancy	Metz	Paris	Reims	Troyes
Nancy	0	55	303	188	183
Metz	55	0	306	176	203
Paris	303	306	0	142	153
Reims	188	176	142	0	123
Troyes	183	203	153	123	0

TABLE 1 – Distances entre les villes, en kilomètres

Le problème se ramène donc à trouver un ordre de visite de ces villes pour lequel la somme des distances est la plus petite possible.

I.2 Algorithme force brute (Méthode exhaustive)

Cette méthode explore systématiquement tous les choix possibles. En effet, ici, on peut facilement calculer toutes les distances parcourues pour tous les itinéraires possibles.
Il existe des symétries entre les itinéraires. . .Par exemple les deux itinéraires suivants sont identiques en terme de distance parcourue (selon le sens de parcours) :

- Nancy → Metz → Paris → Reims → Troyes → Nancy
- Nancy → Troyes → Reims → Paris → Metz → Nancy

Exercice 1 - Le nombre d'itinéraires possibles
En prenant en compte cette remarque, calculer le nombre d'itinéraires possibles pour ce commercial.

Vu le nombre d'itinéraires possibles, nous pouvons facilement les détailler et calculer la distance parcourue pour chacun d'entre eux. L'itinéraire le plus court est

Nancy → Metz → Reims → Paris → Troyes → Nancy

pour une distance totale de 709 km.
Le plus long est

Nancy → Reims → Troyes → Metz → Paris → Nancy

pour une distance totale de 1123 km. Ainsi de manière exhaustive, nous trouvons facilement l'itinéraire le plus court et répondons de façon optimale au problème de ce commercial. Mais il ne visite que quatre villes ...

I.3 Limitation de l'algorithme force brute

Exercice 2 - Plus de villes
Calculer le nombre d'itinéraires possibles si ce commercial a :

- 10 villes à visiter,
- 13 villes à visiter,
- 20 villes à visiter.

Vous pouvez facilement remarquer qu'en se fixant une liste de villes plus longues à visiter, le nombre d'itinéraires possibles devient trop important et ce, même pour les capacités de calcul d'un ordinateur.
Les algorithmiciens qui ont étudié ce problème ont de solides raisons de penser qu'il n'existe aucun algorithme donnant la solution optimale en un temps raisonnable lorsque le nombre de villes est grand. Face à de tels problèmes d'optimisation impossibles à explorer entièrement et exhaustivement, il peut être utile de connaître des algorithmes donnant rapidement une réponse qui, sans être nécessairement optimale, resterait bonne.

I.4 Les algorithmes Gloutons

Les algorithmes gloutons sont utilisés pour répondre à des problèmes d'optimisation, c'est à dire des problèmes algorithmiques dans lesquels l'objectif est de trouver une solution « la meilleure possible » en respectant certaines contraintes parmi un ensemble de solutions.
De nombreuses techniques informatiques sont susceptibles d'apporter une solution exacte ou approchée à ces problèmes d'optimisation. Certaines de ces techniques, comme l'énumération exhaustive de toutes les solutions, ont un coût machine qui les rend souvent peu pertinentes au regard de contraintes extérieures imposées (temps de réponse de la solution, moyens machines limités ...

Définition 1 - Algorithmes gloutons

Les algorithmes gloutons déterminent une solution optimale en effectuant successivement des choix locaux, jamais remis en cause.
Au cours de la construction de la solution, l'algorithme résout une partie du problème puis se focalise ensuite sur le sous-problème restant à résoudre et ainsi de suite.

La question est de savoir si en faisant une série de choix localement optimaux, on finit par aboutir à une solution optimale. C'est parfois le cas mais pas toujours.

I.5 Approche gloutonne sur « Le voyageur de commerce

Appliquons l'approche gloutonne à notre problème du voyageur de commerce. Le problème considéré est la visite de l'ensemble des villes, depuis une ville de départ déterminée et avec retour à cette ville. Une solution est donc n'importe quel itinéraire passant par toutes les villes intermédiaires demandées qu'on peut ramener à la succession de choix : « Quelle sera ma prochaine étape ? » Le critère de choix est ici la distance entre villes. Ces éléments mis en place, il ne reste plus qu'à appliquer la méthode gloutonne : partant de la ville de départ, aller à la ville la plus proche, puis à la ville la plus proche de cette dernière parmi les villes non encore visitées, et ainsi de suite.

Exercice 3 - Solution gloutonne à la main
Trouver l'itinéraire et la distance parcourue en appliquant sur papier cet algorithme glouton à notre commercial.

Vous disposez dans le fichier `TP7.py` d'une liste villes contenant les n villes à visiter numérotées de 0 à $n-1$ en incluant la ville de départ et d'un tableau `dist` à deux dimensions (une liste de listes) représentant les distances entre les villes : `dist[i][j]` donnant la distance entre la ville de numéro i et la ville de numéro j.

Exercice 4 - Villes à enlever
Écrire une fonction `enleve(L, e)` qui prend en paramètre une liste `L` et un élément `e`et renvoie une liste contenant les éléments de `L` à l'exception de `e`. liste L sans l'élément e .

Par exemple, `enleve([7, 5, 9, 3, 5], 3)` donnera `[7, 5, 9, 5]`.

Exercice 5 - Recherche de la ville la plus proche
Écrire une fonction `proche(v_actuelle, v_non_visit)` qui prend en paramètre la ville actuelle `v_actuelle` et la liste des villes à visiter `v_non_visit` et renvoie la ville la plus proche de `v_actuelle` parmi les villes de `v_non_visit` en prenant en compte les distances `dist` entre villes.

Par exemple, la ville la plus proche de Paris est `proche(2, [0, 1, 3, 4]` qui donne 3 : Reims.

Exercice 6 - Glouton
Écrire une fonction `glouton(depart)` qui prend en paramètre le numéro de la ville de départ, `depart` et qui renvoie l'itinéraire sous forme d'une liste ordonnée des numéros des villes à visiter.
En appliquant cette fonction, répondre au problème de notre voyageur de commerce.
Conclure.

Exercice 7 - Distance
Écrire une fonction `distance(itineraire)` qui prend en paramètre un itinéraire sous forme d'une liste ordonnée des villes à visiter et renvoie la distance totale de l'itinéraire.

II Problème 2 : le rendu de monnaie

Ce problème est également un grand classique des algorithmes gloutons, il se résume à comment rendre une somme s donnée avec le minimum de pièces ?
Nous travaillerons avec notre système monétaire comportant des pièces de 200, 100, 50, 20, 10, 5, 2, 1 centimes d'euros (100 et 200 centimes correspondent respectivement aux pièces de 1 et 2 euros). Nous avons donc les valeurs en centimes d'euros suivantes (classées par ordre décroissant) :

```
pieces_euro = [200, 100, 50, 20, 10, 5, 2, 1]
```

Exercice 8
Comment rendre la monnaie de 263 centimes d'euros ?

Exercice 9
Décrire un algorithme glouton pour ce problème.

Exercice 10
Créer une fonction `monnaie(s, pieces)` qui prend en paramètre la somme s à rendre et la liste des pièces possibles (classées par ordre décroissant) et renvoie la liste des nombres de pièces à rendre pour chaque valeur.

Par exemple, `monnaie(191, pieces)` donnera `[0, 1, 1, 2, 0, 0, 0, 1]` qui correspond à une pièce de 100, une pièce de 50, deux pièces de 20 et une pièce de 1.

L'algorithme glouton donne une solution optimale pour le système de pièces proposé.
Ce n'est pas toujours le cas.

Exercice 11 - Penny, shilling et livre
Donner un exemple de somme s pour lequel l'algorithme glouton ne donne pas de solution optimale avec le l'ancien (avant décimalisation) système anglais de pièces :
`[240, 120, 60, 30, 24, 12, 6, 4, 3, 1]`

Les valeurs sont exprimées en pence, un shilling valait 12 pence et une livre valait 20 shillings.

III Problème 3 : le sac à dos

On dispose d'un sac pouvant supporter une masse maximale donnée et de divers objets ayant chacun une valeur et une masse. Il s'agit de choisir les objets à emporter dans le sac afin d'obtenir la valeur totale la plus grande tout en respectant la contrainte de la masse maximale. C'est un problème d'optimisation avec contrainte.
Ce problème peut se résoudre par force brute, c'est-à-dire en testant tous les cas possibles donc de manière exhaustive. Mais ce type de résolution présente un problème d'efficacité. Son coût, en fonction du nombre d'objets disponibles, croît de manière exponentielle.
Un objet est représenté par une liste de la forme `[numero, valeur , masse]` où

- `numero` est un entier, il représente l'objet ; pour simplifier, les objets sont numérotés successivement en commençant par 1,
- `valeur` est un flottant représentant la valeur en euros de l'objet,
- `masse` est un flottant représentant la valeur en kg de l'objet.

Voici deux exemples, représentés par les variables `sac1` et `sac2` dans le fichier `TP7.py`.

Objet	Valeur	Masse
1	2.0	10.0
2	4.0	14.0
3	4.0	7.0
4	5.0	3.0

TABLE 2 – Exemple 1

Objet	Valeur	Masse
1	2.0	10.0
2	4.0	11.0
3	8.0	7.0
4	5.0	5.0
5	8.0	6.0
6	10.0	15.0
7	11.0	8.0

TABLE 3 – Exemple 2

Le principe d'un algorithme glouton est de faire le meilleur choix pour prendre le premier objet, puis le meilleur choix pour prendre le deuxième, et ainsi de suite tout en respectant la contrainte de masse maximale du sac à dos.
Que faut-il entendre par meilleur choix ? Est-ce prendre l'objet qui a la plus grande valeur, l'objet qui a la plus petite masse, l'objet qui a le rapport valeur/masse le plus grand ?
Nous allons tester ces trois stratégies gloutonnes possibles.

Exercice 12
Écrire les fonctions `numero(objet)`, `valeur(objet)`, `masse(objet)` et `rapport(objet)` qui renvoient respectivement le numéro, la valeur, la masse et le rapport valeur/masse de l'objet (un triplet défini ci-dessus).

Par exemple pour `obj1 = sac1[0] = [6, 2.0, 10.0]`
`numero(obj1)` renvoie `6`, `valeur(obj1)` renvoie `2.0`,
`masse(obj1)` renvoie `10.0`, `apport(obj1)` renvoie `0.2`.

Il va être pratique de classer les objets selon la stratégie choisie. Pour cela nous pourrons utiliser la fonction `sorted` ; cette fonction renvoie une liste triée selon le critère et le sens de tri sans modifier l'objet de tri, contrairement à la méthode `.sort()`.

```
def sorted(L, key = None, reverse = False):
```

La fonction optionnelle `key` permet de définir le critère de tri, les éléments successif dans la liste triée en sortie, `L'` vérifieront `key(L'[i]) <= key(L'[i+1])` sauf si la variable optionnelle prend la valeur `True`, dans ce cas on a `key(L'[i]) >= key(L'[i+1])`

- Si on désire classer par ordre décroissant les numéros des objets du sac à dos 1, on écrira

```
L = sorted(sac1, key = numero, reverse = True)
```

- si on désire classer par ordre croissant les valeurs des objets du sac à dos 2,

```
L = sorted(sac2, key = valeur)
```

- si on désire classer par ordre croissant les rapports valeur/masse des objets du sac à dos 2.

```
L=sorted(sac2, key = rapport)
```

On peut alors appliquer une stratégie gloutonne, à partir d'une liste que l'on pourra trier selon le critère désiré.

1. On crée une liste vide, `objets`, destinée à indiquer les objets à garder.
2. On initialise à 0 une variable `masseT`, la masse totale portée par le sac à dos.
3. On initialise à 0 une variable `valeurT`, la valeur totale portée par le sac à dos.
4. On initialise à 0 un indice i, indice dans la liste des objets possibles.
5. Tant qu'il reste des objets à ajouter et que la masse maximale n'est pas atteinte :
   - si la masse de l'objet d'indice i n'est pas trop lourde l'ajouter à `masseT`, ajouter sa valeur de l'objet à `valeurT` et ajouter le numéro de l'objet à `objets`
   - incrémenter i.
6. Renvoyer le triplet `objets, valeurT, masseT`.

Par exemple, en appliquant avec des listes non triées, `glouton(sac1, 30)` donnera `[1, 2, 4], 11.0, 17.0` et `glouton(sac2, 35)` donnera `[1, 2, 3, 4], 19.0, 33.0`.

Exercice 13
Écrire une fonction `glouton(L, masse_max)` qui applique cette stratégie.

Exercice 14
En vous aidant des fonctions précédentes, écrire une fonction `glouton_valeur(sac, masse_max)` qui applique la stratégie du meilleur choix « prendre l'objet qui a la plus grande valeur ».

Exercice 15
En vous aidant des fonctions précédentes, écrire une fonction `glouton_masse(sac, masse_max)` qui applique la stratégie du meilleur choix « prendre l'objet qui a la plus petite masse ».

Exercice 16
En vous aidant des fonctions précédentes, écrire une fonction `glouton_rapport(sac, masse_max)` qui applique la stratégie du meilleur choix « prendre l'objet qui a le rapport valeur/masse le plus grand ».

En appliquant l'algorithme force brute , on trouve les solutions optimales suivantes :

- choix des objets 1, 3 et 4 pour une valeur de 11 euros et une masse de 20 kg dans l'exemple 1 avec un sac de 20 kg,
- choix des objets 3, 5, 6 et 7 pour une valeur de 37 euros et une masse de 36 kg dans l'exemple 2 avec un sac de 40 kg.

Exercice 17
Existe-t-il une stratégie gloutonne optimale ?

IV Problème 4 : maximiser une somme de k éléments parmi n avec contrainte

On cherche à sélectionner k nombres d'une liste à termes positifs en cherchant à avoir leur somme la plus grande possible (maximiser une grandeur) et en s'interdisant de choisir deux nombres voisins (contrainte).
Comme on souhaite avoir le plus grand résultat final, une stratégie gloutonne consiste à choisir à chaque étape le plus grand nombre possible dans les choix restants.

Exercice 18 - Un exemple

```
L = [15,  4, 20, 17, 11,  8, 11, 16,  7, 14,  2,  7,
      5, 17, 19, 18,  4,  5, 13,  8]
```

Appliquez cet algorithme glouton pour 5 éléments sur la liste `L` et calculer la somme totale.
Vérifiez que `[20, 18, 17, 16, 15]` est une autre solution possible.
Que dire de cette stratégie gloutonne ?

Exercice 19
Écrire un programme python qui applique cette stratégie gloutonne.

V Compléments : algorithmes force brute

Le principe est simple : il faut tester tous les cas possibles !
La mise en œuvre l'est moins : comment obtenir tous les cas sans les répéter et sans en oublier un ?

V.1 Problème du sac à dos

Une méthode est d'associer le chiffre 1 à un objet s'il est choisi et le chiffre 0 sinon.
Nous obtenons ainsi un nombre entier écrit en binaire et il suffit d'énumérer toutes les décompositions binaires des entier entre 0 et $2^n - 1$ s'il y a n objets.

Exercice 20
Écrire une fontion `brute_sacados(sac, masse_max)` qui recherche ainsi exhaustivement la solution optimale.

On peut aussi rechercher la solution optimale récursivement en écrivant une fonction `maximum_partiel(liste, k, m)` qui recherche la solution optimale pour la masse `m` parmi les k premiers éléments de la liste.
Si n est la longueur de la liste la solution recherchée sera `maximum_partiel(liste, n, masse_max)`.
Il sera alors possible de diminuer les calculs en mémorisant les résultats intermédiaires, c'est la méthode de programmation dynamique qui sera abordée en spé.

V.2 Problème du maximum sous contrainte

Pour obtenir tous les sous ensembles à k éléments d'une liste donnée, le plus simple est d'utiliser la fonction `combinations` du module `itertools`.

```
from itertools import combinations
```

`combinations(liste, k)` renvoie les combinaisons de longueur k d'éléments pris dans la liste (ou tout objet itérable). On doit ensuite filtrer les combinaisons sans voisins.

Exercice 21
Écrire une fontion `brute_sommeMax(liste, k)` qui recherche exhaustivement la somme maximale de k termes non voisins.

On peut aussi écrire sa propre fonction d'énumération d'indices : en voici une possible

```
def suivant(liste, n):
    k = len(liste)
    i = k - 1
    while i >= 0 and liste[i] == n - k + i:
        i = i - 1
    if i < 0:
        return False
    else:
        liste[i] =liste[i] + 1
        for j in range(i+1, k):
            liste[j] = liste[j-1] + 1
        return True
```

Elle modifie la liste en place et renvoie un booléen pour signifier qu'on a atteint la dernière combinaison. Les combinaisons sont écrites dans l'ordre croissant des termes et l'ordre d'énumération est l'ordre lexicographique.

Ici encore, on peut aussi écrire une fonction récursive qui sera rendue efficace par la programmation dynamique.

Solutions

Solution de l'exercice 1 - Le nombre d'itinéraires possibles
En partant de Nancy et en arrivant à Nancy, vu le nombre de villes à visiter (quatre villes en plus de Nancy), il y a 24 itinéraires possibles (problème de permutation qui correspond au factoriel de 4 : $4! = 1x2x3x4 = 24$). Il existe une symétrie entre ces 24 itinéraires possibles. Ceci nous ramène à $24/2 = 12$ itinéraires possibles.

Solution de l'exercice 2 - Plus de villes
Pour 10 villes à visiter, le nombre d'itinéraires possibles passe à environ 2 millions : $\frac{10!}{2} = 1\,814\,400$.
Pour 13 villes à visiter, on arrive à environ 3 milliards : $\frac{13!}{2} == 3\,113\,510\,400$.
Pour 20 villes à visiter, on dépasse un milliard de milliards : $\frac{20!}{2} = 1\,216\,451\,004\,088\,320\,000$.

Solution de l'exercice 3 - Solution gloutonne à la main
Ici partant de Nancy, nous irons donc en premier lieu à Metz, distante de 55 kilomètres. Ensuite à Reims en 176 kilomètres, puis à Troyes en 123 kilomètres, et enfin à Paris en 153 kilomètres, avec ultime retour à Nancy en 303 kilomètres. On complète ainsi notre itinéraire en 810 kilomètres. Nancy → Metz → Reims → Troyes → Paris → Nancy avec une distance totale de 810 km L'itinéraire ainsi obtenu est plus long que l'itinéraire optimal de 709 kilomètres mais reste loin des assez nombreuses mauvais itinéraires qui demandaient plus de mille kilomètres. Et surtout, nous n'avons analysé qu'un unique itinéraire et ainsi obtenu rapidement une solution.

Solution de l'exercice 4 - Villes à enlever

```
def enleve(L, e):
    '''renvoie une liste correspondant à L sans l'élément e'''
    L_modif = []
    for elt in L :
        if elt != e:
            L_modif.append(elt)
    return L_modif
```

ou encore

```
def enleve(L, e):
    '''renvoie une liste correspondant à L sans l'élément e'''
    i = L.index(e)
    return L[0:i] + L[i+1:]
```

Solution de l'exercice 5 - Recherche de la ville la plus proche

```
def proche(v_actuelle, v_non_visit):
    ''' renvoie la ville la plus proche de la ville actuelle
        selon les distance dist
        et la liste des villes non encore visitées'''
    n = len(v_non_visit) # nombre des villes à visiter
    v_proche = v_non_visit[0] # première ville à visitée
    d_min = dist[v_actuelle][v_proche] # intialisation de la
       distance minimale
    for j in range(1,n): # pour toutes les autres villes
        d = dist[v_actuelle][v_non_visit[j]] # récupération de la
           distance
        if d < d_min: # si on trouve une ville plus proche
            v_proche = v_non_visit[j] # mise à jour de la ville la
                plus proche
            d_min = d # mise à jour de la distance minimale.
    return v_proche
```

Solution de l'exercice 6 - Glouton

```
def glouton(depart):
    itineraire = [depart] # initialisation à la ville de départ
    v_actuelle = depart # ville actuelle , initialisation à la
       ville de départ
    v_non_visit = enleve(villes, depart) # on enlève la ville
       actuelle des villes à visiter
    while len(v_non_visit) != 0: # Tant qu'il y a des villes à
       visiter
        v_proche = proche(v_actuelle, v_non_visit) # recherche de
           la ville la plus proche des villes à visiter
        itineraire.append(v_proche) # ajout dans l'itinéraire
        v_non_visit = enleve(v_non_visit, v_proche) # on enlève la
            ville la plus proche des villes à visiter
        v_actuelle = v_proche # mise à jour de la ville actuelle
           par la ville la plus proche
    itineraire.append(depart) # fin de l'itinéraire avec retour à
       la ville de départ
    return itineraire
```

La solution au problème de notre voyageur de commerce est donnée par glouton(0) qui renvoie [0, 1, 3, 4, 2, 0] correspondant au trajet Nancy → Metz → Reims → Troyes → Paris → Nancy pour une distance de 810 km.
Cette solution n'est pas la solution optimale qui est l'itinéraire Nancy → Metz → Reims → Paris → Troyes → Nancy » pour une distance totale de 709 km. La solution gloutonne donne néanmoins une solution assez proche de la solution optimale.

Solution de l'exercice 7 - Distance

```
def distance(itineraire):
    n = len(itineraire)
    d=0
    for i in range(1,n):
        d = d + dist[itineraire[i-1]][itineraire[i]]
    return d
```

Solution de l'exercice 8
On rend une pièce de 200, de 50, de 10, de 2 et de 1.

Solution de l'exercice 9
Soit s une somme à rendre en centimes d'euros, on donne la plus grande pièce possible parmi toutes les valeurs. Puis à nouveau la plus grande pièce possible pour le montant restant et ainsi de suite.

Solution de l'exercice 10

```
def monnaie(s, pieces):
    n = len(pieces)
    nb_pieces = [0]*n
    reste = s
    for i in range(n):
        nb_pieces[i] = reste // pieces[i]
        reste = reste % pieces[i]
    return nb_pieces
```

Solution de l'exercice 11 - Penny, shilling et livre
Pour rembourser 8 pence, l'algorithme glouton proposera 1 pièce de 6 et 2 pièces de 1, soit 3 pièces alors qu'on peut rembourser avec 2 pièces de 4.

Solution de l'exercice 12

```
def numero(objet):
    ''' Renvoie le numéro de l'objet'''
    return objet[0]

def valeur(objet):
    ''' Renvoie la valeur de l'objet'''
    return objet[1]

def masse(objet):
    ''' Renvoie la masse de l'objet'''
    return objet[2]

def rapport(objet):
    ''' Renvoie le rapport valeur sur masse de l'objet'''
    return objet[1]/objet[2]
```

Solution de l'exercice 13

```
def glouton(L, masse_max):
    objets = [ ] # liste des numéros d'objets à garder
    masse_totale = 0 # masse du sac à dos
    valeur_totale = 0 # valeur du sac à dos
    i = 0 # indice de L
    # Tant qu'il reste des objets à ajouter et que la masse
       maximale n'est pas atteinte
    while masse_totale < masse_max and i < len(L):
        masse_objet = masse(L[i]) # on récupère la masse de l'
           objet i
        if masse_totale + masse_objet <= masse_max: # si elle n'
           est pas trop lourde
            objets.append(numero(L[i])) # on ajoute le numéro de l
               'objet dans la solution
            masse_totale = masse_totale + masse_objet # on calcule
                la masse du sac
            valeur_totale = valeur_totale + valeur(L[i])
        i=i+1 # on passe à l'objet suivant selon le tri effectué
    return objets, valeur_totale, masse_totale
```

Solution de l'exercice 14

```
def glouton_valeur(sac, masse_max):
    # tri par ordre décroissant de la valeur des objets
    L = sorted(sac, key = valeur , reverse= True)
    return glouton(L, masse_max)
```

Solution de l'exercice 15

```
def glouton_masse(sac, masse_max):
    # tri par ordre croissant de la masse des objets
    L = sorted(sac, key = masse )
    return glouton(L, masse_max)
```

Solution de l'exercice 16

```
def glouton_rapport(sac, masse_max):
    L = sorted(sac, key = rapport, reverse=True)
    return glouton(L, masse_max)
```

Solution de l'exercice 17
Aucune stratégie gloutonne ne donne la solution optimale dans tous les cas.

Solution de l'exercice 18 - Un exemple
On prend 20 et on élimine avec ses deux voisins ; il reste

```
[15, 11, 8, 11, 16, 7, 14, 2, 7, 5, 17, 19, 18, 4, 5, 13, 8]
```

On prend 19 et on élimine avec ses deux voisins ; il reste

```
[15, 11, 8, 11, 16, 7, 14, 2, 7, 5, 4, 5, 13, 8]
```

On prend 16 et on élimine avec ses deux voisins ; il reste

```
[15, 11, 8, 14, 2, 7, 5, 4, 5, 13, 8]
```

On prend 15 et on élimine avec son voisin ; il reste

```
[8, 14, 2, 7, 5, 4, 5, 13, 8]
```

On prend 14 et on élimine avec ses deux voisins ; il reste

```
[7, 5, 4, 5, 13, 8]
```

Pour une somme totale de 20+19+16+15+14=84.

On pouvait aussi prendre (dans l'ordre dans la liste, 15, 20, 16, 17 et 18 pour un total de 86. L'algorithme glouton n'est pas optimal.

Solution de l'exercice 19

```
def maximums(L):
    maxi = L[0]
    ind_max = 0
    n = len(L)
    for i in range(n):
        if L[i] > maxi:
            maxi = L[i]
            ind_max = i
    return maxi, ind_max
```

```
def modif(L, ind):
    n = len(L)
    i_min = max(ind - 1, 0)
    i_max = min(n - 1, ind + 1)
    for i in range(i_min, i_max + 1):
        L[i] = 0
```

```
from copy import deepcopy

def somme_max(L, k):
    valeurs = []
    somme = 0
    L_copy = deepcopy(L)
    for i in range(k):
        maxi, ind_maxi = maximums(L_copy)
        valeurs.append(maxi)
        somme += maxi
        modif(L_copy, ind_maxi)
    return valeurs, somme
```

Solution de l'exercice 20

```
def binaire(n, p):
    b2 = [0]*p
    for i in range(p):
        b2[i] = n%2
        n = n//2
    return b2
```

```
def objets(sac, b2):
    n = len(sac)
    val = 0
    mas = 0
    obj = []
    for i in range(n):
        if b2[i] == 1:
            val = val + valeur(sac[i])
            mas = mas + masse(sac[i])
            obj.append(numero(sac[i]))
    return val, mas, obj
```

```
def brute_sacados(sac, masse_max):
    n = len(sac)
    N = 2**n
    v_max = 0
    m_max = 0
    obj_max = []
    for k in range(1, N):
        b2  = binaire(k, n)
        val, mas, obj = objets(sac, b2)
        if val > v_max and mas <= masse_max:
            v_max = val
            m_max = mas
            obj_max = obj
    return obj_max, v_max, m_max
```

Un algorithme récursif peut être

```
def brute_sacados(sac, masse_max):
    n = len(sac)
    def aux(i, m):
        if i == 0:
            return 0, 0, []
        else:
            obj0, v0, m0 = sac[i - 1]
            v1, m1, obj1 = aux(i - 1, m)
            v2, m2, obj2 = aux(i - 1, m - m0)
            if v1 > v2 + v0 or m0 > m:
                return v1, m1, obj1
            else:
                return v0 + v2, m0 + m2, obj2 + [obj0]
    return aux(n, masse_max)
```

Solution de l'exercice 21
On va produire les combinaisons à k éléments de $\{0, 1, \ldots, n-1\}$:

```
for c in combinations(list(range(n)), k)
```

On doit tester la contrainte

```
def test(c):
    p = len(c)
    for i in range(p-1):
        if c[i+1] == c[i] + 1:
            return False
    return True
```

On calcule la somme des termes de `L` pour des indices dans `c`

```
def somme(L, c):
    s = 0
    for k in c:
        s = s + L[k]
    return s
```

On utilise le tout

```
def brute_sommeMax(L, k):
    n = len(L)
    s_max = 0
    c_max = []
    for c in combinations(range(n), k):
        if test(c) and somme(L, c) > s_max:
            c_max = c
            s_max = somme(L, c)
    return s_max, [L[k] for k in c_max]
```

Avec la fonction `suivant`, on peut écrire

```
def brute_sommeMax(L, k):
    n = len(L)
    c = list(range(k))
    s_max = somme(L, c)
    c_max = deepcopy(c)
    while suivant(c, n):
        if test(c) and somme(L, c) > s_max:
            c_max = deepcopy(c)
            s_max = somme(L, c)
    return s_max, [L[k] for k in c_max]
```

Un algorithme récursif peut être

```
def brute_sommeMax(L, k):
    n = len(L)
    def aux(i, p):
        """Somme maximale pour p termes parmi les i premiers"""
        if i < 2*p - 2:
            return "Pas assez de termes"
        elif p == 0:
            return 0, []
        elif i == 2*p -2:
            s1, c1 = aux(i - 2, p - 1)
            return s1 + L[i - 1], c1 + [i - 1]
        else:
            x = L[i-1]
            s1, c1 = aux(i - 2, p - 1)
            s2, c2 = aux(i - 1, p)
            if s1 + x > s2:
                return s1 + x, c1 + [x]
            else:
                return s2, c2
    return aux(n, k)
```