

Programmation dynamique - Problème du sac à dos - Enoncé

Objectif  : Mettre en oeuvre et comparer les stratégies gloutonnes et dynamiques.

Savoir faire  : commenter le code, documenter les fonctions (« Docstring ») et écrire des tests pour les fonctions écrites.

 Respecter les notations de l'énoncé.

Définition du problème

Soit n objets auxquels on associe les valeurs : $v_0, \dots, v_{n-1}$ et les poids $w_0, \dots, w_{n-1}$ (v = value, w = weight).

Le poids maximum du sac à dos est noté W_{max} .

But : maximiser la valeur emportée $v_{max} = \sum_{i \in \{0,1,\dots,n-1\}} v_i$ en respectant la contrainte sur le poids total emporté qui ne doit pas dépasser la capacité du sac : $\sum_{i \in \{0,1,\dots,n-1\}} w_i \leq W_{max}$.

Implémentation - Notations

Les valeurs et les poids ainsi que le poids total du sac sont des entiers strictement positifs.

On fait le choix de travailler exclusivement avec des listes.

On note :

- v la liste de n valeurs comprises entre 1 et une valeur maximale v_m ;
- w la liste de n poids compris entre 1 et une valeur maximale w_m ;
- W_{max} le poids maximum du sac (entier strictement positif).

Dans la suite, on écrit trois fonctions.

1. Un premier algorithme glouton.
2. Un second algorithme glouton utilisant une heuristique différente.
3. Un algorithme dynamique itératif ascendant.
4. Un algorithme dynamique récursif descendant.

Ces fonctions renvoient :

- la valeur maximum déterminée $v_{max} = \sum_{i \in \{0,1,\dots,n-1\}} v_i$;
- les objets choisis dans le cas des algorithmes gloutons ou le tableau des choix optimaux dans le cas de l'algorithme dynamique ;
- le temps de calcul.

Bibliothèques

```
from _validation import *
import random as rd
import matplotlib.pyplot as plt
import matplotlib.lines as lines
from time import perf_counter, sleep
import statistics
```

Rappel python - Module random

module random

`random.seed(a=None)`

Initialise le générateur de nombres aléatoires : en fixant la valeur a , le même tirage aléatoire est obtenu sur différentes machines ou au cours d'exécutions successives.

`random.randint(a, b)`

Renvoie un entier aléatoire N tel que $a \leq N \leq b$.

`random.random()`

Renvoie un nombre aléatoire à virgule flottante x tel que $0.0 \leq x < 1.0$.

`random.uniform(a, b)`

Renvoie un nombre aléatoire à virgule flottante x tel que $a \leq x \leq b$ pour $a < b$ et $b \leq x \leq a$ pour $b < a$.

I Préliminaires

I.1 Fonction tirage

 Ecrire une fonction `tirage(n, vm, wm)` renvoyant deux listes de n éléments.

La première liste est une liste de n valeurs aléatoires comprises entre 1 et la valeur maximum vm .

La seconde liste est une liste de n poids aléatoires compris entre 1 et le poids maximum wm .

Rappel : utiliser `randint(a,b)` de la bibliothèque `random` (chargée ci-dessus sous l'alias `rd`).

I.2 Complément python - `zip()` - "Association" de listes

Objet itérable

Un objet est itérable si on peut parcourir ses éléments à l'aide d'une boucle `for`.

Exemples d'objets itérables : liste, tuple, chaîne, dictionnaire, `range()`...

`zip`

`zip(L1, L2...)` renvoie un objet itérable de tuples de la forme $(L_1[0], L_2[0], \dots), (L_1[i], L_2[i], \dots), \dots$

 Ecrire une fonction `zip2(L1, L2)` effectuant le même travail que la fonction `zip` mais renvoyant une liste (la fonction `zip2` n'utilise donc pas la fonction `zip` de python) .

Quelle est sa complexité en n (longueur des listes `L1` et `L2`) ?

I.3 Rappel - Fonction `lambda` - Définition d'une fonction "à la volée"

Fonction définie "à la volée" = syntaxe courte (sans utiliser `def fct():`)

`f = lambda x: 2*x+1`

Appel de la fonction `f`

`f(3)`

`lambda`

`lambda nom_variable : expression`

I.4 Complément python - sort vs sorted() - Tri de listes

sort et sorted

`liste.sort(key=None, reverse=False)`

`liste.sort()` est une méthode de l'objet liste qui effectue un **tri en place** (liste est modifiée).

`sorted(liste, key=None, reverse=False)`

`sorted(liste)` est une fonction définie sur les listes qui ne modifie pas liste (renvoie une nouvelle liste).

Le paramètre `key` permet de définir un critère de tri à l'aide d'une fonction `lambda`.

I.5 Complément python - perf_counter() - Mesure de durées

La fonction `perf_counter()` du module `time` renvoie un instant (origine des temps inconnue) en secondes.

  Exécuter, observer et commenter le code ci-dessous.

```
ti = perf_counter()    # Instant initial
sleep(2)               # Instructions = code à exécuter (attendre 2 s dans cet exemple)
tf = perf_counter()    # Instant final
Dt = tf - ti           # Durée entre tf et ti
print(f'ti = {ti}, tf = {tf}, tf-ti = {Dt:.3f}') # Durée affichée avec 3 chiffres après la virgule
```

II Stratégie gloutonne

II.1 Fonction glouton1

On donne ci-dessous le code, incomplet, de la fonction `glouton1(v, w, Wmax)`.

 Compléter le code ainsi que les commentaires au sein du code.

```
def glouton1(v, w, Wmax):
    """
    Paramètres :
        v      : listes de valeurs aléatoires pour n=len(v) objets
        w      : listes de poids tirés au hasard pour n objets
        Wmax   : poids maximum du sac à dos

    Renvoie :
        vmax   : valeur maximum déterminée pour le sac à dos
        objets : liste de tuples [(vi, wi),...] = objets choisis
        temps  : durée du calcul en ms
    """
    ti = perf_counter()                #
    n = len(v)                         #
    # Tri de la liste [(vi, wi)] par ordre DECROISSANT sur vi
    v_triee =
    vmax = 0                           #
    Wrestant = Wmax                    # Initialisation du poids RESTANT disponible
    objets = []                        #
    for i in range(n):                 #
        wi = v_triee[i][1]             #
        if wi <= Wrestant:              #
            vi = v_triee[i][0]         #
            vmax += vi                  #
            Wrestant -= wi              #
            objets.append((vi, wi))     #
    tf = perf_counter()                #
    return vmax, objets, round((tf-ti)*1e3, 6) # Durée en ms arrondie à 6 chiffres après la virgule
```

II.2 Questions

 Répondre aux questions en complétant les commentaires ci-dessous.

1. A l'aide du code fourni préciser en quoi consiste la stratégie gloutonne.
2. Pour quelle raison le tri (ligne 16) est-il effectué par ordre décroissant sur les valeurs ?
3. Ligne 28, que représente le troisième item (`round((tf-ti)*1e3, 3)`) renvoyé par la fonction ?

```
# Questions
# 1.
# 2.
# 3.
```

II.3 Fonction glouton2

L'heuristique utilisée dans la fonction `glouton1` n'est pas la meilleure, le critère valeur/poids est a priori beaucoup plus efficace.

 Ecrire le code de la fonction `glouton2(v, w, Wmax)` utilisant cette heuristique (une seule ligne est à modifier!).

III Stratégie dynamique

Algorithme

La mise en oeuvre de la programmation dynamique itérative ascendante repose sur le remplissage d'un tableau de la forme (en prenant l'exemple du cours) :

	0 lbs	1 lbs	2lbs	3lbs	4 lbs
	0	0	0	0	0
Guitar 1lbs \$1500	0	G \$1500	G \$1500	G \$1500	G \$1500
stereo 4lbs \$3000	0	G \$1500	G \$1500	G \$1500	S \$3000
Laptop 3lbs \$2000	0	G \$1500	G \$1500	L \$2000	G+L \$3500

Pour n objets et un sac de capacité maximale W_{max} , le tableau comporte $n + 1$ lignes et $W_{max} + 1$ colonnes. Ce tableau sera nommé v_{max} , représenté par une liste de listes et initialisé par des valeurs nulles.

Notations :

- ligne $i + 1$: évaluation de l'objet de valeur $v_i = v[i]$ et de poids $w_i = w[i]$ à choisir ou non ;
- colonne j : la capacité maximale du sac dans cette colonne est j lbs ;
- $v_{max}[i][j]$ la valeur optimale calculée pour la cellule ligne i colonne j .

Relation de récurrence : $v_{max}[i + 1][j] = \begin{cases} \max(v_{max}[i][j], v[i] + v_{max}[i][j - w[i]]) & \text{si } w[i] \leq j \\ v_{max}[i][j] & \text{si } w[i] > j \end{cases}$

III.1 Approche dynamique itérative ascendante

 Compléter le code de la fonction `dyn_iter(v, w, Wmax)` renvoyant la valeur maximum cherchée, le tableau `choix_opt` ainsi que le temps de calcul en ms.

On pourra utiliser la fonction `max()` de python.

 Ne pas initialiser `vmax` par la commande `[[0] * (Wmax+1)] * (n+1)` (les lignes sont dupliquées et ne sont donc pas indépendantes les unes des autres $\Rightarrow$ comportement inattendu).

```
def dyn_iter(v, w, Wmax):  
    """  
    Paramètres :  
        v      : listes de valeurs aléatoires pour n=len(v) objets  
        w      : listes de poids aléatoires pour n objets  
        Wmax   : poids maximum du sac à dos  
  
    Renvoie :  
        vmax[n][Wmax] : valeur maximum déterminée pour le sac à dos  
        vmax          : Tableau des choix optimaux (liste de listes)  
        temps        : durée du calcul en ms  
    """  
    ti = perf_counter()  
    n = # Nombre d'objets  
    # Initialisation du tableau des choix optimaux n+1 lignes Wmax+1 colonnes  
    vmax = [[0 for j in range(Wmax+1)] for i in range(n+1)]  
    # Remplissage du tableau  
    for i in range(n): # Boucle sur les objets (lignes)  
        for j in range(Wmax+1): # Boucle sur TOUS les poids envisageables (colonnes)  
            if w[i] > j:  
                # ...  
            else:  
                # ...  
    tf = perf_counter()  
    return
```

III.2 Liste des objets choisis

Cette fonction ne fournit pas la liste des objets choisis.

  Pour déterminer les objets choisis, on parcourt le tableau à l'envers en partant de la dernière cellule et on "saute" de choix optimal en choix optimal.
Compléter le code.

```

def objets_choisis(v, w, Wmax, T):
    """
    Paramètres :
        v      : listes de valeurs aléatoires pour n=len(v) objets
        w      : listes de poids aléatoires pour n objets
        Wmax   : poids maximum du sac à dos
        T      : tableau des choix optimaux (liste de listes):

    Renvoie :
        sac : liste des objets choisis : liste de tuples [(v,w), ...]
    """
    sac = [] # Initialisation du sac à dos (liste)
    # On part de la dernière cellule du tableau :
    i = # Nombre d'objets RESTANT à choisir
    j = # Poids RESTANT disponible
    # Parcours du tableau via les choix optimaux :
    while i > 0: # Parcours de la dernière à la première ligne
        if T[i][j] > T[i-1][j]: # L'objet de la cellule (i, j) est optimal
            sac.append( ) # On l'ajoute au sac (attention au décalage des index)
            j -= # Mise à jour du poids restant <=> choix précédent
        i -= # Ligne précédente avec nouveau poids restant
    return sac

```

III.3 Approche dynamique récursive descendante

 Compléter le code de la fonction `dyn_rec(v, w, Wmax)` ci-dessous.

Cette fonction crée un dictionnaire nommé `memo_vmax`, qui remplace le tableau `vmax` de la question III.1, dont les clés sont des tuples (i,j) associés aux valeurs optimales calculées pour la cellule (i,j) du tableau `vmax`.

A l'intérieur de la fonction `dyn_rec`, on définit une fonction récursive `remplissage_memo(i, j)` qui remplit le dictionnaire `memo_vmax` (en utilisant la formule de récurrence ci-dessous dans laquelle l'indice i a été translaté pour tenir compte de la démarche ascendante ($i \rightarrow i - 1$ au lieu de $i \rightarrow i + 1$ pour la démarche descendante)).

On pourra utiliser la fonction `max()` de python.

$$v_{max}[i][j] = \begin{cases} \max(v_{max}[i-1][j], v[i-1] + v_{max}[i-1][j-w[i-1]]) & \text{si } w[i-1] \leq j \\ v_{max}[i-1][j] & \text{si } w[i-1] > j \end{cases}$$

```

def dyn_rec(v, w, Wmax):
    """
    Paramètres :
        v      : listes de valeurs aléatoires pour n=len(v) objets
        w      : listes de poids aléatoires pour n objets
        Wmax   : poids maximum du sac à dos

    Renvoie :
        vmax      : Tableau des choix optimaux (liste de listes)
        vmax[n][Wmax] : valeur maximum déterminée pour le sac à dos
        temps     : durée du calcul en ms
    """
    ti = perf_counter()
    memo_vmax={} # Dictionnaire de la forme {(i, j): valeur} remplaçant le tableau vmax de la question III.1
    def remplissage_memo(i, j):
        if (i,j) not in memo_vmax:
            if i == 0:
                memo_vmax[(i,j)] = v[i-1]
            elif w[i-1] <= j:
                memo_vmax[(i,j)] = max(v[i-1], v[i-1] + memo_vmax[(i-1, j-w[i-1])])
            else:
                memo_vmax[(i,j)] = memo_vmax[(i-1, j)]
        return memo_vmax[(i,j)]
    tf = perf_counter()
    return remplissage_memo(len(v), Wmax), memo_vmax, round((tf-ti)*1e3, 3)

```