

Programmation dynamique - Problème du sac à dos - Corrigé

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`).

```
def tirage(n, vm, wm):  
    """  
    Paramètres :  
        n : nombre d'objets  
        vm : valeur maximum d'un objet  
        wm : poids maximum d'un objet  
    Renvoie :  
        v, w : listes de n valeurs et de n poids tirés au hasard  
    """
```

```
# Test tirage  
tirage(10, 5, 8)
```

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

  Lire attentivement les commentaires, exécuter le code ci-dessous et observer.

```
# On définit deux (ou plus) listes (ou/et tuple) de même longueur  
L1 = ['c', 'a', 'b']  
L2 = [3, 0, 1]  
  
# zip(L1, L2, L3,...) permet d'associer les éléments des différentes listes de même index dans un tuple  
  
for t in zip(L1, L2):    # Parcours par tuple  
    print(t)  
  
for x, y in zip(L1, L2): # Parcours par éléments des tuples  
    print(x, y)  
  
# zip(L1, L2) n'est pas une liste (c'est un objet itérable) mais peut être transformé en liste  
list(zip(L1, L2))
```

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) ?

```
def zip2(L1, L2):
```

```
zip2(L1, L2)
```

La complexité de la fonction est $O(n)$ en raison de la boucle sur les n éléments des deux listes.

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

  Exécuter le code ci-dessous et observer.

```
# Fonction définie "à la volée" = syntaxe courte (sans utiliser def fct():)
f = lambda x: 2*x+1

# Appel de la fonction f
f(3)
```

  Exécuter le code ci-dessous et observer (l'intérêt d'une telle fonction apparaîtra dans la suite).

```
# Fonction renvoyant la 3ème valeur d'une liste ou d'un tuple nommé e
element3 = lambda e : e[2] # Paramètre e = liste ou tuple ; renvoie e[2] = 3ème élément de e

# Appel de la fonction element3
element3([0,1,2,3,4])
```

lambda

lambda nom_variable : expression

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

  Exécuter les deux blocs de code ci-dessous, observer et comparer les deux méthodes de tri.

```
# Tri simple (liste, tuple) par ordre croissant

# Méthode .sort()
L = ['c', 'a', 'b']
L.sort()

# TRI EN PLACE : L est modifiée
L

# Tri simple (liste, tuple) par ordre croissant

# Fonction sorted()
L = ['c', 'a', 'b']

# Création d'une nouvelle liste L n'est pas modifiée
Ltriee = sorted(L)
L, Ltriee
```

```
# Tri simple (liste, tuple) par ordre décroissant
sorted(L, reverse=True)
```

  Exécuter et commenter le code dans la cellule ci-dessous.

```
# Création d'une liste de tuple (on aurait pu utiliser zip comme précédemment)
L = [('c', 2), ('a', 1), ('b', 0)]

# ...(remplacer par un commentaire)...
sorted(L, key=lambda L:L[0])
```

 Ecrire le code permettant d'obtenir la liste [('c', 2), ('a', 1), ('b', 0)] à partir de la liste L.

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 (ligne 16) 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()                # Instant initial (début de l'algorithme)
    n = len(v)                         # Nombre d'objets
    # Tri de la liste [(vi, wi)] par ordre DECROISSANT sur vi
    v_triee =
    vmax = 0                           # Initialisation vmax
    Wrestant = Wmax                    # Initialisation du poids RESTANT disponible
    objets = []                        # Initialisation de la liste des objets choisis
    for i in range(n):                 # Parcours par valeur décroissante (i = objet courant)
        wi = v_triee[i][1]             # Poids de l'objet courant
        if wi <= Wrestant:              # Si on peut choisir l'objet i
            vi = v_triee[i][0]          # On récupère la valeur de l'objet
            vmax += vi                  # On met à jour vmax
            Wrestant -= wi              # On met à jour le poids restant disponible
            objets.append((vi, wi))     # On met à jour la liste des objets
    tf = perf_counter()                # Instant final (fin de l'algorithme)
    return vmax, objets, round((tf-ti)*1e3, 6) # Durée en ms arrondie à 6 chiffres après la virgule

# Test glouton1
v, w = tirage(10, 5, 8)
Wmax = 50
vmax, objets, tcalc = glouton1(v, w, Wmax)
#print(f'Valeurs = {v} \nPoids = {w}')
#print(f'Vmax = {vmax} \nObjets = {objets}')
print(f'Temps de calcul = {tcalc:.3f} ms')
```

Dans cet exemple, le temps de calcul est très faible mais il servira pour des comparaisons dans la suite.

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!).

```
def glouton2(v, w, Wmax):
```

```
# Test glouton2
```

```
vmax, objets, tcalc = glouton2(v, w, Wmax)
```

```
print(f'Valeurs = {v} \nPoids = {w}')
```

```
print(f'Vmax = {vmax} \nObjets = {objets} \nTemps de calcul = {tcalc:.3f} ms')
```

II.4 Comparaison des fonctions glouton1 et glouton2



Exécuter le code à plusieurs reprises en comparant les résultats fournis par les deux algorithmes.

```
# Comparaison glouton1 et glouton2
```

```
v, w = tirage(20, 10, 20)
```

```
Wmax = 40
```

```
vmax1, objets1, tcalc1 = glouton1(v, w, Wmax)
```

```
vmax2, objets2, tcalc2 = glouton2(v, w, Wmax)
```

```
print(f'Valeurs = {v} \nPoids = {w}\n')
```

```
print(f'Glouton 1 \nVmax = {vmax1} \nObjets = {objets1} \nTemps de calcul = {tcalc1:.3f} ms\n')
```

```
print(f'Glouton 2 \nVmax = {vmax2} \nObjets = {objets2} \nTemps de calcul = {tcalc2:.3f} ms')
```

III Stratégie dynamique

Algorithme

La mise en uvre 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 (lignes 14, 16, 18 à 21, 23 et 25) 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           : 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()
    n = len(v) # 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):
        for j in range(Wmax+1):
            # Boucle sur les objets (lignes)
            # Boucle sur TOUS les poids envisageables (colonnes)
            if w[i] > j:
                # Pas d'objet possible
                pass
            else:
                # Calcul de la valeur optimale
                pass

    tf = perf_counter()
    return tf, vmax, choix_opt

# Test dyn_iter avec exemple du cours
v = [1500, 3000, 2000]
w = [1, 4, 3]
Wmax = 4
vmax, tableau, tcalc = dyn_iter(v, w, Wmax)
print(f'Valeurs = {v} \nPoids = {w}')
print(f'Vmax = {vmax} \nTableau = \n{tableau} \nTemps de calcul = {tcalc:.3f} ms')
```

```
# Test dyn_iter avec tirage aléatoire
# Initialisation générateur nombres pseudo-aléatoires : seed(valeur) = > répétabilité
rd.seed(0) # Modifier la valeur ou placer un # devant l'instruction pour obtenir des exécutions différentes
v, w = tirage(10, 5, 5)
Wmax = 8
vmax, tableau, tcalc = dyn_iter(v, w, Wmax)
print(f'Valeurs = {v} \nPoids = {w}')
print(f'Vmax = {vmax} \nTableau = \n{tableau} \nTemps de calcul = {tcalc:.3f} ms')
```

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 (lignes 14, 15, 19, 20 et 21).

```
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 indices)
            j -= # Mise à jour du poids restant <=> choix optimal
        i -= # Ligne précédente avec nouveau poids restant
    return sac

# Test avec exemple du cours
v = [1500, 3000, 2000]
w = [1, 4, 3]
Wmax = 4
vmax, tableau, tcalc = dyn_iter(v, w, Wmax)
print(f'Vmax = {vmax} \nTableau = \n{tableau}')
objets_choisis(v, w, Wmax, tableau)
```

III.3 Approche dynamique récursive descendante

 Compléter le code de la fonction `dyn_rec(v, w, Wmax)` ci-dessous (lignes 18, 20, 22).
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 traduit 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 11
def remplissage_memo(i, j):
    if (i,j) not in memo_vmax:
        if i == 0:
            memo_vmax[(i,j)] =
        elif w[i-1] <= j:

        else:

    return memo_vmax[(i,j)]
tf = perf_counter()
return remplissage_memo(len(v), Wmax), memo_vmax, round((tf-ti)*1e3, 3)

# Test dyn_rec avec exemple du cours
v = [1500, 3000, 2000]
w = [1, 4, 3]
Wmax = 4
vmax, memo_vmax, tcalc = dyn_rec(v, w, Wmax)
print(f'Valeurs = {v} \nPoids = {w}')
print(f'Vmax = {vmax} \nTableau = \n{memo_vmax} \nTemps de calcul = {tcalc:.3f} ms')

```

Décorateur lru_cache (hors programme)

La bibliothèque `functools` de la librairie standard Python fournit un "décorateur" qui enveloppe automatiquement une fonction récursive dans une interface avec un dictionnaire réalisant la mémorisation, sans aucun travail supplémentaire pour le programmeur.

  Exécuter, observer.

L'utilisation du décorateur (ligne 16) permet d'écrire la récurrence sans se soucier de la mémorisation, python la met en place sans autre code nécessaire que `@lru_cache` avant la définition de la fonction décorée.

```

from functools import lru_cache # Bibliothèque dédiée à la mémorisation

def dyn_rec_cache(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
    """

    @lru_cache # la fonction qui suit sera automatiquement mémorisée (=> écriture de la récurrence uniquement)
    def remplissage_memo(i, j):
        if i == 0:
            return 0
        elif w[i-1] <= j:
            return max(remplissage_memo(i-1, j), v[i-1] + remplissage_memo(i-1, j-w[i-1]))
        else:

```

```

        return remplissage_memo(i-1, j)

    return remplissage_memo(len(v), Wmax)

# Test dyn_rec_cache avec exemple du cours
v = [1500, 3000, 2000]
w = [1, 4, 3]
Wmax = 4
vmax = dyn_rec_cache(v, w, Wmax)
print(f'Valeurs = {v} \nPoids = {w}')
print(f'Vmax = {vmax}')

```

IV Etude comparative des algorithmes gloutons et dynamiques

IV.1 Comparaison pour un unique tirage aléatoire

La fonction `comparaison(v, Wmax, vm, wm)` fournie ci-dessous permet de comparer les deux algorithmes gloutons et l'algorithme dynamique itératif ascendant pour un tirage aléatoire.



Exécuter le code afin de pouvoir procéder aux expérimentations ci-dessous.

```

def comparaison(n, v, Wmax, vm, wm, objets=False):
    # Tirage aléatoire des valeurs et des poids
    v, w = tirage(n, vm, wm)
    # Calculs par les 3 algorithmes
    Vmax_g1, objets_g1, tps_g1 = glouton1(v, w, Wmax)
    Vmax_g2, objets_g2, tps_g2 = glouton2(v, w, Wmax)
    Vmax_dy, tableau, tps_dy = dyn_iter(v, w, Wmax)
    objets_dy = objets_choisis(v, w, Wmax, tableau)

    # Valeur max et temps de calcul
    print('
          Valeur max      temps (ms)
    print(f'Glouton 1      | {Vmax_g1} | {tps_g1}')
    print(f'Glouton 2      | {Vmax_g2} | {tps_g2}')
    print(f'Dynamique itérative | {Vmax_dy} | {tps_dy}')

    # Détails - Objets choisis
    if objets:
        print('\n      Objets choisis')
        print('Glouton 2      : ', objets_g2)
        print('Dynamique itérative : ', objets_dy)

    # Comparaisons
    print("\n glouton2 vs dyn_iter :")
    print("  Même valeur max ?           ", Vmax_g2 == Vmax_dy)
    print("  Même nombre d'objets choisis ? ", len(objets_g2) == len(objets_dy))
    print("  Objets choisis identiques ?   ", set(objets_g2) == set(objets_dy))

```



Relancer le code ci-dessous plusieurs fois si nécessaire pour voir apparaître les différences entre les résultats produits (en particulier pour les deux derniers algorithmes).

```

# Nombre d'objets
n = 100
# Poids maximum du sac
Wmax = 300
# Valeur max et poids max pour chaque objet
vm, wm = 20, 30

comparaison(n, v, Wmax, vm, wm)

```

IV.2 Statistiques pour un grand nombre de tirages aléatoires

La fonction `stats(n, Wmax, vm, wm, Ntirages)` fournie ci-dessous permet de comparer les deux algorithmes gloutons et l'algorithme dynamique itératif ascendant en termes de performance (comparaison des valeurs optimales obtenues).

nues) et de complexité temporelle sur un grand nombre de tirages aléatoires.

🖨 Exécuter le code afin de pouvoir procéder aux expérimentations ci-dessous.

```
def stats(n, Wmax, vm, wm, Ntirages):
    # Score = écart relatif entre les valeurs max glouton et valeur max dynamique en %
    scores = [] # Initialisation de la liste des scores

    # Efficacité de glouton2 par rapport à dyn_iter = écart relatif des temps de calcul en %
    tg2 = []
    tdy = []
    efficacite = [] # Initialisation de la liste des temps de calcul relatifs

    for t in range(Ntirages): # Calculs pour N tirages
        v, w = tirage(n, vm, wm) # Tirage valeurs, poids
        Vmax_g2, objets_g2, tps_g2 = glouton2(v, w, Wmax) # Résultats glouton2
        Vmax_dy, tableau, tps_dy = dyn_iter(v, w, Wmax) # Résultats dyn_iter
        scores.append(round((Vmax_dy - Vmax_g2)/Vmax_dy*100, 2)) # Mise à jour liste scores
        tg2.append(tps_g2)
        tdy.append(tps_dy)
    tg2moy = statistics.mean(tg2)
    tg2moy = max(tg2moy, 1e-6) # Pour éviter les divisions par 0 !
    efficacite = [(t-tg2moy)/tg2moy*100 for t in tdy]
    efficacite_moy = statistics.mean(efficacite) # Efficacité moyenne

    # ----- Graphes -----
    plt.clf()
    fig = plt.figure()
    gs = fig.add_gridspec(2, hspace=0.5)
    axs = gs.subplots()
    fig.suptitle('Dynamique vs glouton')

    # Abscisse = tirages
    x = range(Ntirages)

    # Comparaison des complexités temporelles (écart relatif en %)
    axs[0].plot(x, efficacite)
    axs[0].plot([0, Ntirages], [efficacite_moy, efficacite_moy], color='red')
    axs[0].set(xlabel='Tirage', ylabel = 'Efficacité glouton (%)')
    txt = '% moyen = '+str(int(efficacite_moy))
    axs[0].text(10, efficacite_moy*1.05, txt, fontsize='small', color='red', backgroundcolor='white')
    axs[0].set_title("Ecart relatif entre temps d'exécution des algo dynamique et glouton")

    # Comparaison des valeurs optimales (écart relatif en %)
    axs[1].vlines(x, [0]*Ntirages, scores) # Diagramme en bâtons
    axs[1].set(xlabel='Tirage', ylabel = 'Ecart relatif (%)')
    axs[1].set_title('Ecart relatif 0% : valeur max glouton = valeur max dynamique')

    plt.show()
```

💡 Temps de calcul (complexité temporelle) et fiabilité de la version gloutonne.

```
# Nombre d'objets
n = 100
# Poids maximum du sac
Wmax = 300
# Valeur max et poids max pour chaque objet
vm, wm = 20, 30
# Nombre de tirages
Ntirages = 100

stats(n, Wmax, vm, wm, Ntirages)
```

🔪 Que penser de la stratégie gloutonne lorsqu'un écart de quelques % est toléré sur la valeur optimale ?