Partie 1

1. ***On fixe N le nombre d’objets dans le sac à dos. Sans tenir compte de la contrainte de charge maximale, combien de sac à dos contenant N objets peut-on réaliser ? Calculer ce nombre de sacs à dos pour N ∈ {1, 2, 10, 23}.***

Pour déterminer combien de sacs à dos contenant N objets peuvent être réalisés sans tenir compte de la contrainte de charge maximale, il suffit de calculer le nombre de combinaisons de N objets parmi la liste d'objets disponibles. Cela peut être calculé à l'aide de la formule mathématique des combinaisons. Pour N objets, le nombre de combinaisons possibles est donné par "Nombre total d'objets choisis parmi N" :

1. ***Combien d’organisations de sac à dos peut-on former au total, sans tenir compte de la charge maximale de sac à dos ? On inclura les sacs à dos avec les 23 objets, et le sac à dos vide.***

Pour déterminer le nombre total d'organisations de sacs à dos possibles, il faut considérer toutes les combinaisons possibles, y compris le sac à dos vide et celui avec tous les objets. Cela signifie qu'il faut sommer le nombre de combinaisons pour chaque valeur de N, de 0 à 23.

import math

total\_organisations = 0

# Calculer le nombre total d'organisations

for N in range(24):

total\_organisations += math.comb(24, N)

print("Nombre total d'organisations de sacs à dos possibles :", total\_organisations)

1. ***Resoudre le probleme a la main pour C = 0.6.***

Pour résoudre le problème à la main pour C=0.6, nous devons choisir les objets à mettre dans le sac à dos de manière à maximiser la somme des utilités tout en respectant la contrainte de charge maximale.

Voici les étapes pour résoudre ce problème :

* Trier les objets par ordre décroissant de leur rapport d'utilité sur la masse (utilité/masse).
* Commencer à placer les objets dans le sac à dos, en commençant par celui qui a le rapport d'utilité sur la masse le plus élevé.
* Continuer à ajouter des objets jusqu'à ce que la contrainte de charge maximale soit atteinte (la somme des masses des objets dans le sac à dos ne doit pas dépasser C=0.6).
* Noter les objets choisis et leur utilité totale dans le sac à dos.

Je vais trier les objets par rapport d'utilité sur la masse :

Crème Solaire : 1.75/0.4=4.375

Téléphone mobile : 2/0.4=5

Lampes : 1.8/0.3=6

Multi-tool : 1.7/0.2=8.5

Maillon rapide : 1.4/0.05=28

Rustines : 1.5/0.05=30

Pompe : 1.5/0.2=7.5

Démonte-pneus : 1.5/0.1=15

Couteau suisse : 1.5/0.2=7.5

Arrache Manivelle : 0/0.4=0

Clé de 15 : 1/0.3=10/3​≈3.33

Gourde : 2/1=2

Barre de céréales : 0.8/0.4=2

Fruits : 1.3/0.6≈2.167

Désinfectant : 0.6/0.2=3

Pince multiprise : 0.8/0.4=2

Veste de pluie : 1/0.4=2.5

Pantalon de pluie : 0.75/0.4=1.875

Batterie portable : 0.4/0.5​=0.8

Carte IGN : 0.2/0.1=2

Compresses : 0.4/0.1​=4

Chambre à air : 0.5/0.2=2.5

Bouchon valve chromé bleu : 0.1/0.01​=10

Maintenant, commençons à placer les objets dans le sac à dos jusqu'à ce que la contrainte de charge maximale soit atteinte.

~~Lampes (1.8)~~

~~Téléphone mobile (2)~~

~~Crème Solaire (1.75)~~

~~Multi-tool (1.7)~~

~~Rustines (1.5)~~

~~Pompe (1.5)~~

~~La somme des masses est 0.3+0.4+0.4+0.2+0.05+0.2=1.55, ce qui est inférieur à la contrainte de charge maximale C=0.6.~~

~~Donc, les objets choisis sont : Lampes, Téléphone mobile, Crème Solaire, Multi-tool, Rustines et Pompe. L'utilité totale est de 1.8+2+1.75+1.7+1.5+1.5=10.25.~~

1. ***Ecrire un algorithme papier, le plus simple qui vous vienne à l’esprit, pour lequel :***

***• on peut faire varier le nombre d’opérations effectuées par l’algorithme.***

***• dénombrer très facilement le nombre d’opérations (comparaisons, additions, multiplications, ...) que l’algorithme effectue pour se terminer.***

***• l’algorithme se termine.***

Algorithme SacADosSimple(liste\_objets, capacité):

1. Initialiser un sac à dos vide.

2. Trier la liste d'objets par rapport d'utilité sur la masse (utilité/masse) de manière décroissante.

3. Pour chaque objet dans la liste triée :

a. Si l'objet peut être ajouté au sac sans dépasser la capacité :

i. Ajouter l'objet au sac.

ii. Mettre à jour la capacité restante.

b. Sinon :

i. Passer à l'objet suivant.

4. Retourner le sac à dos avec les objets sélectionnés.

Algorithme SacADosSimple2(liste\_objets, capacité):

1. Initialiser un sac à dos vide.

2. Générer toutes les combinaisons possibles d'objets à partir de la liste d'objets.

3. Pour chaque combinaison de la liste :

a. Vérifier si la somme des masses des objets de la combinaison est inférieure ou égale à la capacité maximale :

i. Si oui, calculer l'utilité totale de la combinaison.

b. Mettre à jour la combinaison optimale en fonction de l'utilité totale maximale trouvée jusqu'à présent.

4. Retourner la combinaison optimale de sac à dos avec les objets sélectionnés.

1. ***Ecrire ce programme sous Python. A l’aide de la librairie time déterminer le temps de calcul de votre CPU pour réaliser n opérations. On fera varier n afin d’obtenir une idée du temps de réponse de l’algorithme en fonction du nombre d’opérations à effectuer. On notera T la durée (moyenne estimée) en secondes de la réalisation d’une opération.***

import time

class Item:

def \_\_init\_\_(self, name, mass, utility):

self.name = name

self.mass = mass

self.utility = utility

def sac\_a\_dos\_simple(liste\_objets, capacite):

sac = []

capacite\_restante = capacite

debut = time.time()

# Trie des objets par rapport d'utilité sur la masse

liste\_objets.sort(key=lambda x: x.utility / x.mass, reverse=True)

# Parcours de la liste d'objets

for objet in liste\_objets:

# Vérification de la capacité restante

if objet.mass <= capacite\_restante:

# Ajout de l'objet au sac

sac.append(objet)

# Mise à jour de la capacité restante

capacite\_restante -= objet.mass

fin = time.time()

temps\_execution = fin - debut

return sac, temps\_execution

# Exemple d'utilisation

if \_\_name\_\_ == "\_\_main\_\_":

# Création de la liste d'objets

liste\_objets = [

Item("Pompe", 0.2, 1.5),

Item("Démonte-pneus", 0.1, 1.5),

Item("Gourde", 1, 2),

# Ajoutez les autres objets ici

]

# Capacité du sac à dos

capacite = 0.6

# Nombre d'opérations à effectuer

n = len(liste\_objets)

# Exécution de l'algorithme avec différentes tailles de liste

for i in range(1, n+1):

sous\_liste = liste\_objets[:i]

sac, temps\_execution = sac\_a\_dos\_simple(sous\_liste, capacite)

print(f"Nombre d'objets : {i}, Temps d'exécution : {temps\_execution} secondes.")

1. ***A l’aide des approximations ci-dessus, et en supposant que chaque opération possède le même coût en temps, combien de temps faudrait-il pour tester toutes les organisations de sac à dos (dans le cas où il n’y a pas de contrainte sur la masse) ?***

Pour estimer le temps nécessaire pour tester toutes les organisations de sacs à dos dans le cas où il n'y a pas de contrainte sur la masse, nous pouvons utiliser l'approche suivante :

Calculer le nombre total d'organisations de sacs à dos possibles, ce que nous avons déjà fait précédemment.

Estimer le temps moyen d'une opération en exécutant l'algorithme pour un petit ensemble d'objets et en mesurant le temps d'exécution.

Multiplier le nombre total d'organisations par le temps moyen d'une opération pour estimer le temps total nécessaire.

Supposons que nous ayons déjà calculé le nombre total d'organisations de sacs à dos possibles et que nous ayons mesuré le temps moyen d'une opération T.

Alors, le temps total nécessaire serait approximativement :

Temps total=Nombre total d’organisations×T

Ainsi, si N est le nombre total d'organisations et T est le temps moyen d'une opération, le temps total nécessaire pour tester toutes les organisations serait N×T.

1. ***Rédiger un algorithme papier pour la résolution exacte du problème du sac à dos. On l’appellera A.***

Algorithme A (SacADosExact):

1. Initialiser une matrice M de taille (nombre d'objets + 1) x (capacité maximale + 1) à zéro.

2. Pour chaque objet i de 1 à n (nombre d'objets) :

Pour chaque capacité j de 1 à C (capacité maximale) :

Si le poids de l'objet i est inférieur ou égal à la capacité j :

M[i][j] = maximum de (utilité de l'objet i + M[i-1][j-poids de l'objet i], M[i-1][j])

Sinon :

M[i][j] = M[i-1][j]

3. Retourner la valeur de M[n][C], qui représente l'utilité maximale atteignable avec les objets et la capacité données.

Rq : Cet algorithme utilise une approche itérative pour remplir une matrice M où chaque case M[i][j] représente l'utilité maximale atteignable avec les i premiers objets et une capacité maximale de j. La solution optimale est alors trouvée dans la case M[n][C], où n est le nombre total d'objets et C est la capacité maximale du sac à dos.

1. ***Rédiger cet algorithme exact A sous Python.***
2. ***A l’aide de A, déterminer la (ou les) composition(s) du sac à dos dans le cas ou C = 2, C = 3, C = 4 et C = 5. On précisera le temps de calcul pour chaque valeur de C. Gardez ces valeurs pour les challenges temps et précision.***

import time

class Item:

def \_\_init\_\_(self, name, mass, utility):

self.name = name

self.mass = mass

self.utility = utility

def sac\_a\_dos\_exact(liste\_objets, capacite):

n = len(liste\_objets)

capacite\_max = int(capacite \* 10) # Convertir la capacité en entier pour l'indexation de la matrice

M = [[0] \* (capacite\_max + 1) for \_ in range(n + 1)]

debut = time.time()

# Remplissage de la matrice M

for i in range(1, n + 1):

for j in range(1, capacite\_max + 1):

if liste\_objets[i - 1].mass <= j:

M[i][j] = max(liste\_objets[i - 1].utility + M[i - 1][j - liste\_objets[i - 1].mass], M[i - 1][j])

else:

M[i][j] = M[i - 1][j]

# Reconstruction de la solution

sac = []

utilite\_max = M[n][capacite\_max]

j = capacite\_max

for i in range(n, 0, -1):

if M[i][j] != M[i - 1][j]:

sac.append(liste\_objets[i - 1])

j -= int(liste\_objets[i - 1].mass \* 10)

fin = time.time()

temps\_execution = fin - debut

return sac, utilite\_max, temps\_execution

# Exemple d'utilisation

if \_\_name\_\_ == "\_\_main\_\_":

# Création de la liste d'objets

liste\_objets = [

Item("Pompe", 0.2, 1.5),

Item("Démonte-pneus", 0.1, 1.5),

Item("Gourde", 1, 2),

# Ajoutez les autres objets ici

]

# Capacités du sac à dos à tester

capacites = [2, 3, 4, 5]

# Test de l'algorithme pour chaque capacité

for capacite in capacites:

sac, utilite\_max, temps\_execution = sac\_a\_dos\_exact(liste\_objets, capacite)

print(f"Pour C = {capacite}, Composition du sac à dos : {[(obj.name, obj.mass, obj.utility) for obj in sac]}")

print(f"Utilité maximale : {utilite\_max}, Temps d'exécution : {temps\_execution} secondes.")

1. ***Rédiger à présent un algorithme papier heuristique pour la résolution du problème du sac à dos. On l’appellera B. La base de données de ce sac à dos est disponible sous Moodle.***

Algorithme B (SacADosHeuristique) :

1. Trier la liste d'objets par ordre décroissant de leur utilité.

2. Initialiser un sac à dos vide.

3. Pour chaque objet dans la liste triée :

Si l'objet peut être ajouté au sac sans dépasser la capacité maximale :

Ajouter l'objet au sac.

Sinon :

Passer à l'objet suivant.

4. Retourner le sac à dos avec les objets sélectionnés.

Rq : Cet algorithme utilise une heuristique simple qui consiste à choisir d'abord les objets ayant la plus grande utilité. Ensuite, il les ajoute au sac à dos tant que la capacité maximale n'est pas dépassée. Cela ne garantit pas une solution optimale, mais peut donner de bons résultats dans de nombreux cas.

1. ***A l’aide de B, déterminer la (ou les) composition(s) du sac à dos dans le cas où C = 2, C = 3, C = 4 et C = 5. On précisera le temps de calcul pour chaque valeur de C. Gardez ces valeurs pour les challenges temps et précision.***

import time

class Item:

def \_\_init\_\_(self, name, mass, utility):

self.name = name

self.mass = mass

self.utility = utility

def sac\_a\_dos\_approche(liste\_objets, capacite):

sac = []

capacite\_restante = capacite

debut = time.time()

# Trie des objets par ordre décroissant d'utilité

liste\_objets.sort(key=lambda x: x.utility, reverse=True)

# Parcours de la liste d'objets

for objet in liste\_objets:

# Vérification de la capacité restante

if objet.mass <= capacite\_restante:

# Ajout de l'objet au sac

sac.append(objet)

# Mise à jour de la capacité restante

capacite\_restante -= objet.mass

fin = time.time()

temps\_execution = fin - debut

return sac, temps\_execution

# Exemple d'utilisation

if \_\_name\_\_ == "\_\_main\_\_":

# Création de la liste d'objets

liste\_objets = [

Item("Pompe", 0.2, 1.5),

Item("Démonte-pneus", 0.1, 1.5),

Item("Gourde", 1, 2),

# Ajoutez les autres objets ici

]

# Capacités du sac à dos à tester

capacites = [2, 3, 4, 5]

# Test de l'algorithme pour chaque capacité

for capacite in capacites:

sac, temps\_execution = sac\_a\_dos\_approche(liste\_objets, capacite)

print(f"Pour C = {capacite}, Composition du sac à dos : {[(obj.name, obj.mass, obj.utility) for obj in sac]}")

print(f"Temps d'exécution : {temps\_execution} secondes.")

1. ***Comparer A et B.***
2. ***Ecrire un nouvel algorithme B′ approché renvoyant une solution approchée à ce problème en une durée inférieure à 2 secondes. On comparera cette solution approchée à la solution exacte obtenue précédemment. Gardez ces valeurs pour les challenges temps et précision.***

Comparaison entre les algorithmes A (exact) et B (heuristique) :

Exactitude :

* A (Exact) : Fournit toujours la solution optimale.
* B (Heuristique) : Fournit une solution qui peut être sous-optimale.

Temps d'exécution :

* A (Exact) : Le temps d'exécution peut être significativement plus long, surtout pour de grands ensembles d'objets et des capacités élevées.
* B (Heuristique) : Le temps d'exécution est généralement plus court car l'algorithme est plus simple et ne nécessite pas de calculs intensifs.

Complexité :

* A (Exact) : A une complexité temporelle et spatiale élevée, généralement exponentielle, en fonction du nombre d'objets et de la capacité.
* B (Heuristique) : A une complexité temporelle et spatiale généralement linéaire ou légèrement supérieure, car il ne nécessite qu'un tri et une itération simple sur la liste des objets.

Fiabilité :

* A (Exact) : Fournit toujours une solution correcte et optimale.
* B (Heuristique) : La qualité de la solution dépend de la qualité de l'heuristique et peut varier selon les cas.

Pour un cas où la précision est primordiale et où le temps d'exécution n'est pas un problème, l'algorithme A est préférable car il garantit une solution optimale. Cependant, si le temps d'exécution est critique ou si la solution n'a pas besoin d'être optimale, l'algorithme B peut être plus approprié en raison de sa rapidité d'exécution.

Pour un nouvel algorithme B' approché qui doit fournir une solution en moins de 2 secondes, une possibilité serait d'utiliser une heuristique plus sophistiquée, comme une approche gloutonne améliorée ou une métaheuristique telle que la recherche locale ou l'algorithme génétique. Ces approches peuvent fournir des solutions de qualité acceptable dans des délais stricts. Je pourrais élaborer sur une approche particulière si cela vous intéresse !

Test2+3 projetmaths.py : Le premier code utilise une approche de recherche exhaustive pour trouver la meilleure combinaison d'articles dans un sac à dos en respectant une contrainte de masse maximale. Il génère toutes les combinaisons possibles d'articles et sélectionne celle qui maximise l'utilité totale sans dépasser la masse maximale autorisée.

Test5+6+7 projetmaths.py : Le deuxième code utilise une approche gloutonne suivie d'une optimisation. Dans la première étape, il trie les articles en fonction de leur score (rapport utilité/masse) dans l'ordre décroissant, puis sélectionne autant d'articles que possible en commençant par les plus performants jusqu'à ce que la masse maximale soit atteinte. Ensuite, il optimise cette sélection initiale en explorant toutes les combinaisons possibles des articles restants pour trouver celle qui maximise l'utilité totale sans dépasser la masse maximale autorisée.

En résumé, le premier code effectue une recherche exhaustive pour trouver la meilleure solution possible, tandis que le deuxième code utilise une approche gloutonne suivie d'une optimisation pour trouver une solution de bonne qualité mais pas nécessairement optimale. La différence principale réside dans la façon dont les combinaisons d'articles sont générées et évaluées pour trouver la meilleure solution.

Test8projetmaths.py: partie II SNCF sans IHM

Test9projetmaths.py: partie II SNCF avec IHM

<https://fr.wikipedia.org/wiki/Heuristique_(math%C3%A9matiques)>

<https://stackoverflow.com/questions/54106071/how-can-i-set-up-a-virtual-environment-for-python-in-visual-studio-code>

<https://stackoverflow.com/questions/56658553/why-do-i-get-a-modulenotfounderror-in-vs-code-despite-the-fact-that-i-already>

<https://stackoverflow.com/questions/42463866/how-to-use-pip-with-visual-studio-code>

<https://code.visualstudio.com/docs/python/python-quick-start>

php server

<https://stackoverflow.com/questions/68739105/read-excel-file-with-two-headers-as-a-dataframe-and-generate-a-new-header>

<https://stackoverflow.com/questions/60879718/how-to-use-pandas-to-only-read-excel-header>

<https://www.listendata.com/2017/02/import-data-in-python.html>

<https://docs.github.com/fr/codespaces/developing-in-a-codespace/using-github-codespaces-in-visual-studio-code>

La complexité de cet algorithme peut être analysée en plusieurs étapes, en commençant par l'analyse de la fonction combinations et en continuant avec l'utilisation de cette fonction dans find\_best\_combination.

**Analyse de la fonction combinations**

La fonction combinations génère toutes les combinaisons possibles d'une liste d'items. La récursivité de cette fonction implique qu'elle explore toutes les sous-listes possibles, y compris l'ensemble vide. La complexité de cette fonction peut être décrite comme suit :

* Pour une liste de longueur nnn, il y a 2n2^n2n combinaisons possibles (chaque élément peut être présent ou non).
* Ainsi, la complexité en temps de la génération de toutes les combinaisons est O(2n)O(2^n)O(2n).

**Analyse de la fonction find\_best\_combination**

La fonction find\_best\_combination utilise la fonction combinations pour obtenir toutes les combinaisons possibles des items. Ensuite, elle évalue chaque combinaison pour vérifier si elle respecte la contrainte de masse maximale et si elle maximise l'utilité.

Voici l'analyse détaillée :

1. **Génération des combinaisons** :
   * La génération des combinaisons prend O(2n)O(2^n)O(2n) temps comme discuté précédemment.
2. **Évaluation des combinaisons** :
   * Chaque combinaison est évaluée pour sa masse totale et son utilité totale.
   * Pour chaque combinaison, la somme des masses et des utilités est calculée. Si on suppose que chaque combinaison a en moyenne kkk éléments, cela prendra O(k)O(k)O(k) temps.
   * Cependant, dans le pire des cas, kkk peut être proche de nnn, donc chaque évaluation pourrait prendre O(n)O(n)O(n) temps.
3. **Total des évaluations** :
   * Il y a 2n2^n2n combinaisons à évaluer, et chaque évaluation prend O(n)O(n)O(n) temps.
   * Donc, le temps total pour évaluer toutes les combinaisons est O(n⋅2n)O(n \cdot 2^n)O(n⋅2n).

**Conclusion**

L'algorithme find\_best\_combination a une complexité exponentielle due à la génération et l'évaluation de toutes les combinaisons possibles d'items. La complexité en temps est O(n⋅2n)O(n \cdot 2^n)O(n⋅2n).

Cela signifie que cet algorithme est inefficace pour de grandes valeurs de nnn (nombre d'items) et pourrait devenir impraticable au-delà de quelques dizaines d'items. Pour résoudre des problèmes de sac à dos plus grands, il est souvent nécessaire d'utiliser des approches plus sophistiquées comme la programmation dynamique ou des heuristiques.