PROJET 7 : RÉSOLVEZ DES PROBLÈMES EN UTILISANT DES ALGORITHMES EN PYTHON

INTRODUCTION :

Nous devons résoudre un problème knapsack classique en utilisant les deux approches suivantes :

- Un algorithme « Brute force » qui passe sur toutes les données
- Un algorithme optimisé basé sur la programmation dynamique

DÉTAILS DU PROBLÈME

Objectif :

Choisir l'ensemble d'actions le plus rentable parmi une liste en respectant un montant de dépenses maximum défini.

Contraintes :

- Chaque action ne peut être achetée qu'une seule fois
- Le portefeuille maximum est fixé à 500€
- Les actions ne sont pas divisibles

Solutions :

- Un algorithme "Bruteforce" avec une approche naïve
- Un algorithme "optimisé" avec une approche en programmation dynamique

ALGORITHME BRUTE-FORCE

- La solution brute force consiste à essayer toutes les combinaisons d'actions possibles et de choisir la plus rentable.
- Cette solution est très gourmande en temps et extrêmement sensible à l'augmentation de N (nombre d'éléments passés à l'algorithme)

PSEUDO-CODE BRUTE-FORCE

```
brute_force(actions, W):
        n = len(actions)
        for i in range (n):
                for combination in combinations(actions, i):
                        compare combination rentability
```

ANALYSE BRUTE-FORCE

- **Analyse temporelle :**

L'algorithme Brute-Force itère n (nombre d'éléments passés dans l'algorithme) fois sur la fonction itertools.combinaison dont la complexité temporelle est O(n!)

La complexité temporelle est donc n*n! soit O(n!) selon la notation BigO

- **Analyse de la mémoire :**

L'algorithme crée n liste de complexité spatiale O(n), sa complexité spatiale est donc de O(n^2).

RÉSULTATS BRUTE-FORCE

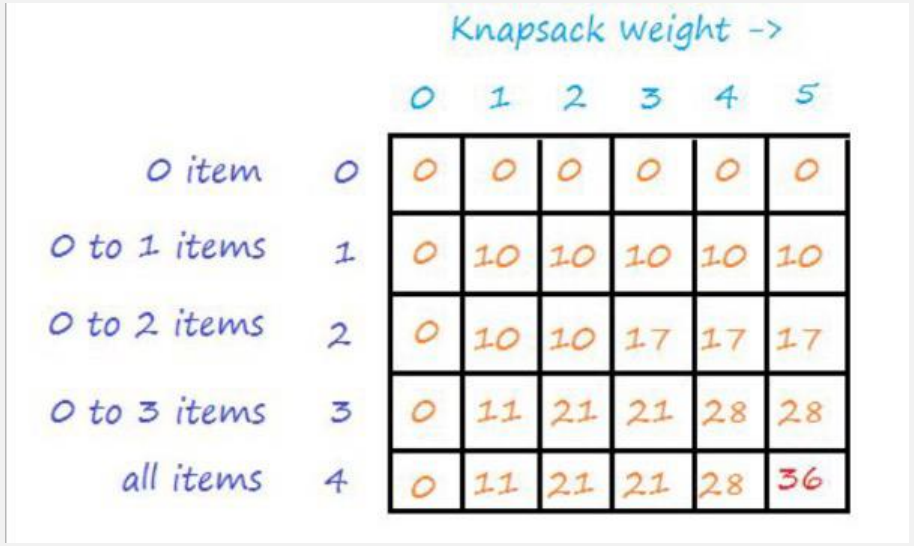
```
Résultat bruteforce :
La combinaison optimale est (('Action-4', 70.0, 14.0), ('Action-5', 60.0, 10.200000000000001), ('Action-6', 80.0, 20.0), ('Action-8', 26.0, 2.86), ('Action-10', 34.0, 9.18), ('Action-11', 42.0, 7.140000000000001),
('Action-13', 38.0, 8.74), ('Action-18', 10.0, 1.4000000000000001), ('Action-19', 24.0, 5.04), ('Action-20', 114.0, 20.52))
Le profit maximum est de 99.08€ pour un investissement est de 498.0€
Execution time : 1.4039123058319092 seconds
```

Le total des profits est de 99,08€ après deux ans
Le prix total est de 498€
L'algorithme met 1,4 secondes à trouver la solution pour ce premier dataset.

ALGORITHME OPTIMISÉ

- La solution optimisée s'appuie sur le principe de la programmation dynamique.
- L'algorithme va créer une table pour stocker la solution de chaque sous problème rencontré.

Cette table lui permet de ne pas effectuer plusieurs fois les mêmes calculs (ou sous problèmes) et de gagner un temps considérable par rapport à une approche naïve.



		Knapsack weight ->					
		0	1	2	3	4	5
0 item	0	0	0	0	0	0	0
0 to 1 items	1	0	10	10	10	10	10
0 to 2 items	2	0	10	10	17	17	17
0 to 3 items	3	0	11	21	21	28	28
all items	4	0	11	21	21	28	36

PSEUDO-CODE ALGORITHME OPTIMISÉ

- Création d'une matrice pour recevoir le résultats des sous problèmes :

```
matrice = [[0 for x in range(max_cost + 1)] for x in range(len(actions) + 1)]
```

- Peuplement de la matrice en résolvant les sous problèmes :

```
for i in range(1, len(actions) + 1):
        for j in range(1, max_cost + 1):
            if actions[i-1][1] <= j:
                matrice[i][j] = max(actions[i-1][2] + matrice[i-1][j-actions[i-1][1]], matrice[i-1][j])
            else:
                matrice[i][j] = matrice[i-1][j]
```

ANALYSE ALGORITHME OPTIMISÉ

- **Analyse temporelle :**

L'algorithme optimisé crée une matrice (ou table) de n (nombre d'items passés en paramètres) par w (capacité, ici appelée portefeuille). Il itère ensuite sur toutes les cellules de cette matrice (soit n*w cellules).

La complexité temporelle est donc O(n*w) selon la notation BigO.

- **Analyse de la mémoire :**

L'algorithme crée une matrice en deux dimensions de taille n*w. Sa complexité spatiale est donc O(n*w).

COMPARAISON BRUTE-FORCE/OPTIMISÉ

Sur le dataset 0, les deux algorithmes fournissent les valeurs suivantes :

Brute force :

Prix total : 498,0€

Bénéfices après deux ans : 99,08€

Temps d'exécution ≃ 1,4sec

Algorithme optimisé:

Prix total : 498,0€

Bénéfices après deux ans : 99,08€

Temps d'exécution ≃ 0,004 sec

- On observe que les deux algorithmes trouvent les mêmes résultats mais le second est presque 3 fois plus rapide sur ce dataset.

COMPARAISON AVEC SIENNA DATASET 1

Mes résultats :

- La combinaison optimale est ['Share-DBUJ', 'Share-KMTG', 'Share-GHIZ', 'Share-NHWA', 'Share-UEZB', 'Share-LPDM', 'Share-MTLR', 'Share-USSR', 'Share-GTQK', 'Share-FKJW', 'Share-MLGM', 'Share-CGJM', 'Share-WPLI', 'Share-LGWG', 'Share-ZSDE', 'Share-SKKC', 'Share-QQTU', 'Share-GIAJ', 'Share-CBNY', 'Share-XJMO', 'Share-LRBZ', 'Share-KZBL', 'Share-EMOV', 'Share-IFCP']
- Le profit maximum est de **198.53€** pour un investissement de 500€
- Execution time : 1.478 seconds

Résultats de Sienna
Action achetée : Share-GRUT
Prix total : 498.76€
Bénéfices après deux ans : **196.61€**

COMPARAISON AVEC SIENNA DATASET 2

Mes résultats :

- La combinaison optimale est ['Share-ECAQ', 'Share-IXCI', 'Share-FWBE', 'Share-ZOFA', 'Share-PLLK', 'Share-ANFX', 'Share-PATS', 'Share-VCXT', 'Share-NDKR', 'Share-ALIY', 'Share-JWGF', 'Share-JGTW', 'Share-FAPS', 'Share-LFXB', 'Share-DWSK', 'Share-UPCV', 'Share-XQII', 'Share-ROOM']
- Le profit maximum est de **198.04€** pour un investissement de 500€
- Execution time : 0.831 seconds

Résultats de Sienna :

Actions achetées : Share-ECAQ, Share-IXCI, Share-FWBE, Share-ZOFA, Share-PLLK, Share-YFVZ, Share-ANFX, Share-PATS, Share-NDKR, Share-ALIY, Share-JWGF, Share-JGTW, Share-FAPS, Share-VCAX, Share-LFXB, Share-DWSK, Share-XQII, Share ROOM

Prix total : 489.24€

Bénéfices après deux ans : **193.78€**