Projet 7 :Résolvez des problèmes en utilisant des algorithmes en Python

Présenté par : Abdelwahid HADJ ZOUBIR

Mentor : Alexandre Iwanesko

Sommaire

- Contexte du projet
- Processus suivi
- Partie 1 : Algorithme « brute force »
  - Analyse de l'algorithme
  - Démonstration du programme
- Partie 2 : Algorithme optimisé, Problème « KnapSack »
  - Principe de résolution : « Dynamic programming »
  - Analyse de l'algorithme : Complexité temporelle & spéciale
  - Analyse de l'algorithme : Limites
  - Démonstration du programme
  - Comparaison des résultats de l'algorithme optimisé et les résultats de Sienna
- Discussion
- Débriefing

Contexte du Projet :

Parcours DA-Python :

- Projet n°7 du parcours
- Résoudre des problèmes à l'aide des algorithmes

Scénario :

- Développeur chez « AlgoInvest&Trade »
  - Coder un algorithme « brute force » permettant de trouver la meilleurs solution possibles parmi un ensemble
  - Coder un algorithme « optimisé » permettant de faire la même chose que précédemment plus rapidement

Enjeu : Permettre à **AlgoInvest&Trade** de conduire les meilleurs opérations possibles pour leurs clients

Processus suivi :

Identification du problème:

- Identification d'un ensemble de N actions à effectuer (acheter)
- Chaque actions peut être effectué d'une fois (acheté qu'une seule fois)
- Chaque action possède un coût « w » et une valeur « v » (un bénéfice qui est un pourcentage du coût)
- Trouver un ensemble d'actions à effectuer, dans le but d'avoir la plus grande sommes possible des valeurs des actions dans cet ensemble, Sans que la somme des coût ne dépasse un maximum de « W »

Algorithme « Brute force » :

- Tester toutes les combinaisons possibles des actions et trouver l'ensemble d'action qui répond aux critères énoncés

Algorithme « Optimisé » :

- Trouver de manière intelligente la meilleurs combinaison possible d'actions qui répond aux critères énoncés sans tester toutes les combinaisons possibles

Partie 1 : Algorithme « Brute force »

Un algorithme de type « brute force » est un algorithme qui teste toutes les solution possibles avant de fournir la meilleure qui est recherchée

- Pour notre problème il existe : $\sum_{k=1}^{k=n} \frac{n!}{k!(n-k)!}$ combinaisons possibles d'actions, sans répétition (une action ne peut apparaitre d'une seule fois dans une combinaisons)
- Tester chaque combinaisons vis-à-vis des critères du problème
- Mettre toutes les combinaisons répondants aux critères dans une liste
- Sortir la combinaison qui obtient le meilleurs résultat

Partie 1 : Algorithme « Brute force »
Analyse de l'algorithme

Il existe une liste d'actions de taille N

List_action = [action_1, action_2……., action_N]

Liste contenant toutes les combinaisons vérifiées : list_combi = []

Le coût d'une action « i » : action_i.cout

Le bénéfice d'une action « i » : action_i.benefice

Ainsi nous testons toutes les combinaison possible en faisant une analyse temporelle grâce à la notation BigO

Partie 1 : Algorithme « Brute force »
Analyse de l'algorithme

```
List_action = [action_1, action_2, ……, action_N] →n * O(1)

Somme_benef = 0 →n * O(1)

Somme_cout = 0 →n * O(1)

Somme_cout_max = 500 →n * O(1)

Combi = 0 →n * O(1)

list_combi = [] →n * O(1)

for action in list_action:

    combi = combi + action →n * O(1)
        if (action.cout < somme_cout_max) and (action.benefice >= somme_benefice): →n * O(1)
            list_combi.append(action) →n * O(1)


    for action in (list_action - 1):

        combi = combi + action →n * O(1)
            if (action.cout < somme_cout_max) and (action.benefice >= somme_benefice): →n * O(1)
                list_combi.append(action) →n * O(1)
```

N

N - 1

Partie 1 : Algorithme « Brute force »
Analyse de l'algorithme

```
for action in (list_action – 2):
        combi = combi + action →n * O(1)
            if (action.cout < somme_cout_max) and (action.benefice >= somme_benefice): →n * O(1)
                list_combi.append(action) →n * O(1)
```

N - 2

```
for action in (list_action – (n-1)):
        combi = combi + action →n * O(1)
            if (action.cout < somme_cout_max) and (action.benefice >= somme_benefice): →n * O(1)
                list_combi.append(action) →n * O(1)
```

N – (N - 1)

combi_gagnante = max(list_combi) →n * O(1)

Partie 1 : Algorithme « Brute force »
Analyse de l'algorithme

La complexité temporelle T(n) est ensuite calculée comme suit :

$T(n) = n*O(1) + 5*O(1) + (3n*O(1)) * (4(n-1)*O(1)) * (4*(n-2)*O(1)) * (4*(n-3)*O(1)) * ….. * (4*(2)*O(1)) * (4*(1)*O(1))$

$T(n) = n + (3n) * 4(n-1) * 4*(n-2) * 4*(n-3) * 4*(n-4) *……..* 4*(n-k) * 4*(3) * 4*(2) * 4*(1)$

$T(n) = n + n * (n-1) * (n-2) * (n-3) * (n-k) * 3 * 2 * 1$

$T(n) = n * (n-1) * (n-2) * (n-3) * (n-k) * 3 * 2 * 1$

$T(n) = n!$

$T(n) = O(n!)$ (Complexité factorielle)

Partie 1 : Algorithme « Brute force » Démonstration du programme

Partie 2 : Algorithme « Optimisé »

L'algorithme « optimisé » permet de trouver de manière intelligente la meilleurs solution possible et le plus rapidement possible

- Ensemble d'actions de taille N et coût maximal W
- A chaque action « i » une valeur « Vi » et un coût « Wi »
- Sortir la combinaison qui obtient le meilleurs résultat sans considérer toutes les possibilité
- Garantie une résolution « rapide » et précise du problème

Partie 2 : Algorithme « Optimisé »

Principe de résolution : « Dynamic Programming »

Le principe de cette méthode est de diviser le problème en plusieurs sous problèmes. Dans notre cas :

- Pour un coût maximal de W à ne pas dépasser, nous considérons plusieurs sous problèmes ou le coût maximal vas de 0 jusqu'à W (0, 1, 2,……, W)
- Et pour chaque sous problème on considère de manière évolutive l'ensemble des actions qui est pris en compte (action 1, action 1 & 2,….., action 1 & 2 &……& N)
- Cela permet de construire un tableaux « DP » de 2 Dimension (DP (N,W))
- Ce tableau sera remplis de manière progressive avec la somme des valeurs des actions pour arriver à la somme maximal contenue dans le dernier élément du tableau
- Deux cas de figures : A chaque étape, une action peut soit être choisi soit ignoré

Partie 2 : Algorithme « Optimisé »
Principe de résolution : « Dynamic Programming »

Ainsi pour un nombre N d'action et pour un critère de coût maximal de W:

- Nous construisons un tableau de taille N*W (N lignes et W colonnes) qui contiendra les sommes des valeurs des actions choisis.
- Pour chaque étape si le coût maximal = 0 alors la somme des valeur = 0 (aucune action ne peut être choisi), Aucune valeur n'est ajouté pour s'il n y a pas d'action (1ère ligne et 1ère colonne contiennent des 0)
- Pour chaque action, en commençant par l'action 1, et pour chaque sous problème pour un coût maximal allant de 1 jusqu'à W, nous décidons si une action sera choisi ou non tout en tenant compte des action choisis précédemment
- Enfin une fois le tableau remplis, la somme maximale des valeurs des actions est contenue dans le dernier élément du tableau
- Il suffit de reconstituer de manière récursive la liste des actions qui ont contribuées à la solution

Partie 2 : Algorithme « Optimisé »
Principe de résolution : « Dynamic Programming »

```
Pour i de 0 à N
        Pour i de 0 à N
                element_tableau = 0
Pour i de 1 à N
        Pour i de 1 à N
                element(i, j) = maximun de ( element(i-1, j) ou Valeur Vi +
                element (i-1, j - coût de l'action i (Wi)) si j >= Wi
Pour i de N à 0
        si action_i choisi alors
                list_action,append(action_i)
                somme_valeurs = somme_valeurs - valeur action_i
        sinon continuer
        si somme_valeurs = 0
                arret du programme
```

Partie 2 : Algorithme « Optimisé »

Analyse de l'algorithme : Complexité temporelle & spéciale

Pour i de 0 à N → N* O(1)
    Pour i de 0 à W → W* O(1)
        element_tableau = 0 → O(1)
Pour i de 1 à N → N* O(1)
    Pour j de 1 à W → W* O(1)
        element(i, j) = maximun de (element(i-1, j) ou Valeur Vi +
        element (i-1, j - coût de l'action i (Wi)) si j >= Wi →O(1)
Pour i de N à 0 →N* O(1)
    si action_i choisi alors
        list_action,append(action_i) → O(1)
        somme_valeurs = somme_valeurs - valeur action_i → O(1)
    sinon continuer
    si somme_valeurs = 0
        arret du programme → O(1)

Partie 2 : Algorithme « Optimisé »

Analyse de l'algorithme : Complexité temporelle & spéciale

Complexité spatiale : Celle-ci n'est calculé que pour ce qui est enregistré en mémoire.

S(n) = N*W (lors de la construction du tableau DP(N, W) et qui est à ce moment la enregistré en mémoire)

Complexité temporelle :

Dans ce cas :

T(N) = N*O(1) * W*O(1) + N*O(1) * W*O(1) + N*O(1)

T(N) = 2 * N*W*O(1) + N*O(1)

T(N) = N*W*O(1) + N*O(1)

T(N) = N*W*O(1)

T(N) = O(N*W)

Partie 2 : Algorithme « Optimisé »
Analyse de l'algorithme : Complexité temporelle & spéciale

Complexité spatiale S(N) = O(N*W)

Complexité temporelle T(N) = O(N*W)

Partie 2 : Algorithme « Optimisé »
Analyse de l'algorithme : Limites

Limites théorique :

La complexité temporelle de notre algorithme O(N*W) est une complexité linéaire et cela veut dire que le nombre d'opérations nécessaires à la résolution di problème évolue proportionnellement et de manière linéaire à N par un facteur W qui peut être fixe.

De ce fait théoriquement une machine sera toujours capable de résoudre notre problème même si N est très grand.

Limite pratique :

Considérant l'aspect pratique, la puissance d'un processeur et la taille de la RAM fixe une limite à la convergence de notre algorithme. Mais cette limite reste tout de même très élevée

Partie 2 : Algorithme « Optimisé »
Analyse de l'algorithme : Limites

Dans notre cas :

N*W = 50.000.000

Un processeur d'une puissance de calcule de 2Ghz pour effectuer 20 million d'opération par seconde

Aussi Pour une mémoire RAM de 8 Gigabytes nous pouvons enregistrer 40 Milliards de résultats.

Partie 2 : Algorithme « Optimisé » :Comparaison des résultats ensemble de données 1

Résultat Algo Optimisé	Résultats Sienna
Bénéfice Total : 198.54 euros	Bénéfice Total : 196.61 euros
Coût total : 499.95 euros	Coût total : 498.76 euros
Liste des actions :	Liste des actions :
'name': 'Share-KMTG', 'price': 23.21, 'profit_euro': 9.28	'name': 'Share-GRUT', 'price': 498.76, 'profit_euro': 196,61
'name': 'Share-GHIZ', 'price': 28.0, 'profit_euro': 11.17	
'name': 'Share-NHWA', 'price': 29.18, 'profit_euro': 11.6	
'name': 'Share-UEZB', 'price': 24.87, 'profit_euro': 9.81	
'name': 'Share-LPDM', 'price': 39.35, 'profit_euro': 15.63	
'name': 'Share-MTLR', 'price': 16.48, 'profit_euro': 6.59	
'name': 'Share-USSR', 'price': 25.62, 'profit_euro': 10.14	
'name': 'Share-GTQK', 'price': 15.4, 'profit_euro': 6.15	
'name': 'Share-FKJW', 'price': 21.08, 'profit_euro': 8.39	
'name': 'Share-MLGM', 'price': 0.01, 'profit_euro': 0.0	
'name': 'Share-QLMK', 'price': 17.38, 'profit_euro': 6.86	
'name': 'Share-WPLI', 'price': 34.64, 'profit_euro': 13.82	
'name': 'Share-LGWG', 'price': 31.41, 'profit_euro': 12.41	
'name': 'Share-ZSDE', 'price': 15.11, 'profit_euro': 6.03	
'name': 'Share-SKKC', 'price': 24.87, 'profit_euro': 9.82	
'name': 'Share-QQTU', 'price': 33.19, 'profit_euro': 13.14	
'name': 'Share-GIAJ', 'price': 10.75, 'profit_euro': 4.29	
'name': 'Share-XJMO', 'price': 9.39, 'profit_euro': 3.75	
'name': 'Share-LRBZ', 'price': 32.9, 'profit_euro': 13.14	
'name': 'Share-KZBL', 'price': 28.99, 'profit_euro': 11.35	
'name': 'Share-EMOV', 'price': 8.89, 'profit_euro': 3.51	
'name': 'Share-IFCP', 'price': 29.23, 'profit_euro': 11.66	

Partie 2 : Algorithme « Optimisé » :Comparaison des résultats ensemble de données 2

Résultat Algo Optimisé	Résultats Sienna
Bénéfice Total : 197.95 euros	Bénéfice Total : 193.78 euros
Coût total : 499.9 euros	Coût total : 489.24 euros
Liste des actions :	Liste des actions :
'name': 'Share-ECAQ', 'price': 31.66, 'profit_euro': 12.5	'name': 'Share-ECAQ', 'price': 31.66, 'profit_euro': 12.5
'name': 'Share-IXCI', 'price': 26.32, 'profit_euro': 10.37	'name': 'Share-IXCI', 'price': 26.32, 'profit_euro': 10.37
'name': 'Share-FWBE', 'price': 18.3, 'profit_euro': 7.29	'name': 'Share-FWBE', 'price': 18.3, 'profit_euro': 7.29
'name': 'Share-ZOFA', 'price': 25.32, 'profit_euro': 10.07	'name': 'Share-ZOFA', 'price': 25.32, 'profit_euro': 10.07
'name': 'Share-PLLK', 'price': 19.94, 'profit_euro': 7.96	'name': 'Share-PLLK', 'price': 19.94, 'profit_euro': 7.96
'name': 'Share-LXZU', 'price': 4.24, 'profit_euro': 1.68	'name': 'Share-YFVZ', 'price': 22.55, 'profit_euro': 8.82
'name': 'Share-YFVZ', 'price': 22.55, 'profit_euro': 8.82	'name': 'Share-ANFX', 'price': 38.54, 'profit_euro': 15.31
'name': 'Share-ANFX', 'price': 38.54, 'profit_euro': 15.31	'name': 'Share-PATS', 'price': 27.7, 'profit_euro': 11.07
'name': 'Share-PATS', 'price': 27.7, 'profit_euro': 11.07	'name': 'Share-NDKR', 'price': 33.06, 'profit_euro': 13.19
'name': 'Share-SCWM', 'price': 6.42, 'profit_euro': 2.45	'name': 'Share-ALIY', 'price': 29.08, 'profit_euro': 11.61
'name': 'Share-NDKR', 'price': 33.06, 'profit_euro': 13.19	'name': 'Share-JWGF', 'price': 48.69, 'profit_euro': 19.44
'name': 'Share-ALIY', 'price': 29.08, 'profit_euro': 11.61	'name': 'Share-JGTW', 'price': 35.29, 'profit_euro': 13.91
'name': 'Share-JWGF', 'price': 48.69, 'profit_euro': 19.44	'name': 'Share-FAPS', 'price': 32.57, 'profit_euro': 12.88
'name': 'Share-JGTW', 'price': 35.29, 'profit_euro': 13.91	'name': 'Share-VCAX', 'price': 27.42, 'profit_euro': 10.69
'name': 'Share-FAPS', 'price': 32.57, 'profit_euro': 12.88	'name': 'Share-LFXB', 'price': 14.83, 'profit_euro': 5.9
'name': 'Share-VCAX', 'price': 27.42, 'profit_euro': 10.69	'name': 'Share-DWSK', 'price': 29.49, 'profit_euro': 11.6
'name': 'Share-LFXB', 'price': 14.83, 'profit_euro': 5.9	'name': 'Share-XQII', 'price': 13.42, 'profit_euro': 5.30
'name': 'Share-DWSK', 'price': 29.49, 'profit_euro': 11.6	'name': 'Share-ROOM', 'price': 15.06, 'profit_euro': 5.90
'name': 'Share-XQII', 'price': 13.42, 'profit_euro': 5.3	
'name': 'Share-ROOM', 'price': 15.06, 'profit_euro': 5.91	

Remarque:

- Dans le cas de notre algorithme, les coûts doivent être exprimé en entiers
- De ce fait il a fallu tout multiplier par 100
- Cela à eu pour effet de multiplier par 100 le nombre de donnés à traiter
- Les donnée de départ N*W = N*500 ~= 500.000 opérations et données sauvegardées
- Pour faire fonctionner l'algorithme il a fallu faire N*W*100 = N*500*100 ~= 50.000.000 opérations et données sauvegardées

Merci !

Discussion

Débriefing