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
- 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 - 1

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.

Remarque:

Dans le cas de notre algorithme, les coûts doivent être exprimé en entiers

Merci !

Discussion

Débriefing