Résolvez des problèmes en utilisant des algorithmes en Python

Magali Courté
Développeuse d'application Python
Projet 7

Contexte du projet

- But du projet :
  Création d'algorithmes permettant de calculer la meilleure combinaison d'actions (rapport coût / profit avec un budget maximal),
- Algorithme Bruteforce :
  Calculer le coût et le profit de chaque combinaison possible et ne garder que la meilleure,
- Algorithme Optimized :
  Déterminer la combinaison la meilleure directement en calculant son coût et son profit en temps réel.

Objectifs et contraintes

- **Objectif :**
  Maximiser le profit des clients après deux ans d'investissement,

- **Contraintes :**
  Chaque action ne peut être achetée qu'une fois, et le montant maximal de dépense par client est de 500 euros,

- **Présentation des données:**
  Une liste d'actions avec les coûts par action et les bénéfices potentiels après deux ans d'investissement.

Pseudo – Code Bruteforce

```python
def generate_combinations(actions_updated):
    n = len(actions_updated)
    all_combinations = []
    for i in range(1, 2 ** n):
        combination = []
        for j in range(0, n):
            if (i >> j) & 1:
                combination.append(actions_updated[j])

        all_combinations.append(combination)

    print("Le nombre total de combinaisons possibles est: ", len(all_combinations))
    return all_combinations
```

Analyse Bruteforce

Fonction generate_combinations(actions_updated) :

La fonction génère toutes les combinaisons possibles.
On a le choix, à chaque fois, de « prendre » ou « ne pas prendre » une action, n fois. Nous avons donc 2*2*2*…..(n fois)

Complexité temporelle : **O(2^n * n)**, où n est le nombre d'actions.

Complexité Spatiale : **O(2^n)**, pour stocker toutes les combinaisons possibles.

Fonction profit_cout_combinaison(combinaison) :

La fonction effectue une somme des coûts et profits de chaque action dans une combinaison.
La complexité dépend du nombre d'actions dans la combinaison.
Si on note n le nombre d'action totale, on peut noter m le nombre d'action dans une combinaison (m est donc une fraction de n)

Complexité temporelle : **O(m)**

Fonction meilleur_profit(all_combinaison, budget_max) :

La fonction recherche la meilleure combinaison en respectant la contrainte du budget maximal.
La complexité dépend du nombre total de combinaisons et du nombre d'actions dans chaque combinaison.

Complexité temporelle : **O(2^n * m)**

La complexité temporelle de l'algorithme correspond à la complexité la plus défavorable.
Nous avons donc, dans notre cas : O(2^n*m).

Résultats Bruteforce

La meilleure combinaison est la suivante:
['**Action-4**', '**Action-5**', '**Action-6**', '**Action-8**', '**Action-10**', '**Action-11**', '**Action-13**', '**Action-18**', '**Action-19**', '**Action-20**']

Le profit total de cette combinaison est: **99.08 €**
Le coût total d'achat des actions de la combinaison est: **498.0 €**
Utilisation de la mémoire : **197.33 MiB**

Durée d'exécution du programme: **04s:419ms**

Optimized

<u>Programmation dynamique :</u>
Un problème se décompose en sous problèmes.
Ces sous-problèmes sont résolus de manière récursive ou itérative.
Pour éviter de recalculer plusieurs fois les mêmes sous-problèmes on stocke les résultats intermédiaire dans une table.

Pseudo – Code Optimized

```
def dynamique_best_combinaisons(actions_updated, budget_max):
    actions_list = list(actions_updated)
    n = len(actions_list)
    dp = [[0 for _ in range(budget_max + 1)] for _ in range(n + 1)]
    selected_actions = [[[] for _ in range(budget_max + 1)] for _ in range(n + 1)]

    for i in range(0, n):
        action = actions_list[i]
        action_price = int(action.get('price'))
        action_profit = int(action.get('profit'))

        for j in range(1, budget_max):
            if action_price <= j:
                if action_profit + dp[i][j - action_price] > dp[i][j]:
                    dp[i + 1][j] = action_profit + dp[i][j - action_price]
                    selected_actions[i + 1][j] = selected_actions[i][j - action_price] +
[action.get("name")]
                else:
                    dp[i + 1][j] = dp[i][j]
                    selected_actions[i + 1][j] = selected_actions[i][j]
            else:
                dp[i + 1][j] = dp[i][j]
                selected_actions[i + 1][j] = selected_actions[i][j]

    combinaison_optimale = selected_actions[n][budget_max - 1]
    return combinaison_optimale
```

Analyse Optimized

dynamique_best_combinaisons(actions_updated, budget_max) :

Complexité temporelle : **O(n * budget_max)**, où n est le nombre d'actions.
Complexité Spatiale : **O(n * budget_max)**, pour les listes dp et selected_actions.

profit_cout_combinaison(actions_list, budget_max) :

Complexité temporelle : **O(n)**, où n est le nombre d'actions.
Complexité Spatiale : **O(n * budget_max)**, pour les listes dp et selected_actions créées dans dynamique_best_combinaisons.

Au global, la complexité est de l'ordre de **O(n * budget_max)** . Elle est linéaire par rapport au produit du nombre d'actions (n) et du budget maximal (budget_max),
Si le nombre d'actions ou le budget maximal augmente, le temps d'exécution augmentera de manière.

Résultats Optimized

La meilleure combinaison est la suivante:
['Action-4', 'Action-5', 'Action-6', 'Action-8', 'Action-10', 'Action-11', 'Action-13', 'Action-18', 'Action-19', 'Action-20']

Le profit total de cette combinaison est: **99.08 €**
Le coût total d'achat des actions de la combinaison est: **498.0 €**
Utilisation de la mémoire : **18.84 MiB**

Durée d’exécution du programme : **0s654ms**

Comparaison

La meilleure combinaison est la suivante dans les deux cas:
['Action-4', 'Action-5', 'Action-6', 'Action-8', 'Action-10', 'Action-11', 'Action-13', 'Action-18', 'Action-19', 'Action-20']
Le profit total de cette combinaison est: **99.08 €**
Le coût total d'achat des actions de la combinaison est: **498.0 €**

	Bruteforce.py	Optimized.py
Utilisation de la mémoire	197.33 MiB	18.84 MiB
Temps d'exécution du programme	4 s 419 ms	0 s 654 ms

Comparaison

- Avantages
- Inconvénients

Comparaison Set 1

- Sienna : algorithme Glouton
- Temps d'exécution : 34s:904ms
- Utilisation de la mémoire : 15.07 MiB

- Comparaison des ratios :
  - Optimized.py = 198.55 / 499.96 = 0.397
  - Algo Sienna = 196.71 / 498.76 = 0.0394

Algorithme Optimized			
Name	**Price**	**Benefice %**	**Profit**
Share-IFCP	29,23 €	39,88	11,66 €
Share-EMOV	8,89 €	39,52	3,51 €
Share-KZBL	28,99 €	39,14	11,35 €
Share-LRBZ	32,90 €	39,95	13,14 €
Share-XJMO	9,39 €	39,98	3,75 €
Share-GIAJ	10,75 €	39,90	4,29 €
Share-QQTU	33,19 €	39,60	13,14 €
Share-SKKC	24,87 €	39,49	9,82 €
Share-ZSDE	15,11 €	39,88	6,03 €
Share-LGWG	31,41 €	39,50	12,41 €
Share-WPLI	34,64 €	39,91	13,82 €
Share-QLMK	17,38 €	39,49	6,86 €
Share-MLGM	0,01 €	18,86	0,00 €
Share-FKJW	21,08 €	39,78	8,39 €
Share-GTQK	15,40 €	39,95	6,15 €
Share-USSR	25,62 €	39,56	10,14 €
Share-MTLR	16,49 €	39,97	6,59 €
Share-LPDM	39,35 €	39,73	15,63 €
Share-UEZB	24,87 €	39,43	9,81 €
Share-NHWA	29,18 €	39,77	11,60 €
Share-GHIZ	28,00 €	39,89	11,17 €
Share-KMTG	23,21 €	39,97	9,28 €

Total Cost **499,96 €**
Total Profit **198,55 €**

Sienna Choice			
Name	**Price**	**Benefice %**	**Profit**
Share-GRUT	498,76 €	39,42	196,61 €

Total Cost **498,76 €**
Total Profit **196,61 €**

Différence entre les résultats	
Cost	**1,20 €**
Profit	**1,94 €**

Magali Courté

Comparaison Set 2

- Temps d'exécution Optimized : 24s:867ms
- Utilisation de la mémoire Optimized : 17.25 MiB

Algorithme Optimized			
Name	**Price**	**Benefice %**	**Profit**
Share-ROOM	15,06 €	39,23	5,91 €
Share-XQII	13,42 €	39,51	5,30 €
Share-DWSK	29,49 €	39,35	11,60 €
Share-LFXB	14,89 €	39,79	5,92 €
Share-VCAX	27,42 €	38,99	10,69 €
Share-FAPS	32,57 €	39,54	12,88 €
Share-JGTW	35,29 €	39,43	13,91 €
Share-JWGF	48,69 €	39,93	19,44 €
Share-ALIY	29,08 €	39,93	11,61 €
Share-NDKR	33,06 €	39,91	13,19 €
Share-SCWM	6,42 €	38,10	2,45 €
Share-PATS	27,70 €	39,87	11,04 €
Share-ANFX	38,55 €	39,72	15,31 €
Share-YFVZ	22,55 €	39,10	8,82 €
Share-LXZU	4,24 €	39,54	1,68 €
Share-PLLK	19,94 €	39,91	7,96 €
Share-ZOFA	25,32 €	39,78	10,07 €
Share-FWBE	18,31 €	39,82	7,29 €
Share-IXCI	26,32 €	39,40	10,37 €
Share-ECAQ	31,66 €	39,49	12,50 €
Total Cost	**499,98 €**		
Total Profit			**197,96 €**

Sienna Choice			
Name	**Price**	**Benefice %**	**Profit**
Share-ROOM	15,06 €	39,23	5,91 €
Share-XQII	13,42 €	39,51	5,30 €
Share-DWSK	29,49 €	39,35	11,60 €
Share-LFXB	14,89 €	39,79	5,92 €
Share-VCAX	27,42 €	38,99	10,69 €
Share-FAPS	32,57 €	39,54	12,88 €
Share-JGTW	35,29 €	39,43	13,91 €
Share-JWGF	48,69 €	39,93	19,44 €
Share-ALIY	29,08 €	39,93	11,61 €
Share-NDKR	33,06 €	39,91	13,19 €
Share-PATS	27,70 €	39,87	11,04 €
Share-ANFX	38,55 €	39,72	15,31 €
Share-YFVZ	22,55 €	39,10	8,82 €
Share-PLLK	19,94 €	39,91	7,96 €
Share-ZOFA	25,32 €	39,78	10,07 €
Share-FWBE	18,31 €	39,82	7,29 €
Share-IXCI	26,32 €	39,40	10,37 €
Share-ECAQ	31,66 €	39,49	12,50 €
Total Cost	**489,24 €**		
Total Profit			**193,78 €**

- Comparaison des ratios :
  - Optimized.py = 197.96 / 499.98 = 0.3959
  - Algo Sienna = 193.78 / 489.24 = 0.3960

Différence entre les résultats	
Cost	**10,74 €**
Profit	**4,18 €**

Analyse Résultats

Set 1 : Le ratio bénéfice/dépense est plus avantageux avec notre algorithme

Set 2 : Le ratio bénéfice/dépense est plus avantageux dans la solution de Sienna. Mais on respecte la consigne d'être le plus proche des 500 euros d'investissement du client.

A prendre en compte :
Quel est le temps d'exécution du programme de Sienna et combien de mémoire utilise-t'il? Ces données doit impérativement être prise en compte pour une véritable analyse.

Axes d’améliorations

- Éviter de copier/coller les données de base dans un nouveau .csv . Il vaut mieux prendre un fichier de base, et ajouter à la suite les colonnes avec les datas transformées.
- Utilisation de Dataframe de Pandas pour éviter certaines boucles for, notamment :

```
for action in actions_list:
    if float(action['price']) > 0 and float(action['profit']) > 0:
       action['profit'] = ((float(action['price'])) * (((float(action['profit'])))) / 100)"
```


Magali Courté

Axes d’améliorations

- Utilisation de la fonction itertools pour calculer toutes les combinaisons possibles : itertools.combinations(iterable, r)

```python
def generate_combinations(actions_updated):
    all_combinations = []
    n = len(actions_updated)
    # Générer toutes les combinaisons possibles de longueur 1 à n
    for r in range(1, n + 1):
        # Utiliser itertools.combinations pour générer les combinaisons de longueur r
        for combination in itertools.combinations(actions_updated, r):
            all_combinations.append(list(combination))
```