Algorithme de force brute

Analyse

Cette démarche consiste à **chercher et rassembler toutes les solutions,** pour ensuite évaluer la plus optimisée à partir de toutes les solutions trouvées.
Pour cela, on s'oriente vers l'utilisation d'une **fonction récursive**, qu'on va nécessairement commencer par un **point d'arrêt**, condition if/else dans notre cas.
Cependant, il y a un très grand nombre de solutions à parcourir et le programme peut prendre un très **grand nombre de temps** pour toutes les calculer.
Cela va donc nous amener à ensuite s'orienter vers une solution optimisée en termes de temps afin de réduire le temps de calcul de la solution cherchée. ⇒ importance de la solution optimisée
Variables de départ ci-dessous :

```
liste_actions = [('action_1', 20, 5), ('action_2', 30, 10), ('action_3', 50, 15), ('action_4', 70, 20),
                 ('action_5', 60, 17), ('action_6', 80, 25), ('action_7', 22, 7), ('action_8', 26, 11),
                 ('action_9', 48, 13), ('action_10', 34, 27), ('action_11', 42, 17), ('action_12', 110, 9),
                 ('action_13', 38, 23), ('action_14', 14, 1), ('action_15', 18, 3), ('action_16', 8, 8),
                 ('action_17', 4, 12), ('action_18', 10, 14), ('action_29', 24, 21), ('action_20', 114, 18)]
actions_selectionnees = []
benefice_total = 0
cout_total = 0
plafond = 500
```

Algorithme de force brute

Pseudocode

```
actions_selectionnees ← []
benefice_total ← 0
cout_total ← 0
plafond ← 500

fonction force_brute(plafond, liste_actions, actions_selectionnees) :
        si liste_actions n'est pas vide :
                val1, lstVal1 ← force_brute(plafond, liste actions[excepte_1ere], actions_selectionnees)
                val ← liste_actions[1ere]
                si val[cout] <= plafond :
                        val2, lstVal2 ← force_brute(plafond, liste_actions[excepte_1ere], actions_selectionnees + [val])
                        si val < val2 :
                                retourner val2, lstVal2
                        retourner val1, lstVal1
                sinon :
                        retourner(somme(actions_selectionnees[benefice]), afficher(actions_selectionnees))
```

Solution optimisée

Analyse

Approche

- **tri décroissant** des actions selon leur taux de rentabilité
- parcours des éléments et sélection des actions tant qu'assez de budget (**plafond non atteint**)

Intérêt : Plus rapide que la solution force brute
Limitations : Résultats non optimales (mettre un exemple où c'est pas le bon)
Dans le cas où nous avons 100€ de budget :

- action_1 : 60€, 11% = 6,6€
- action_2 : 50€, 10% = 5€
- action_3 : 45€, 10% = 4,5€

Glouton ⇒ la solution est (action1 toute seule = 6,6€)
Optimale ⇒ la solution est (action_2, action_3 = 9,5€) c'est mieux parce qu'on utilise mieux tout le budget, et on peut ainsi obtenir un **meilleur bénéfice** (11,1€ au lieu de 6,6€).

Solution optimisée

Pseudocode

```
actions_selectionnees ← []
benefice_total ← 0
cout_total ← 0
plafond ← 500

fonction methode_glouton(plafond, liste_actions, cout_total, actions_selectionnees) :
        actions_triees ← trier selon le coût liste_actions par ordre décroissant
        actions_selectionnees ← []
        tant que action_triees n'est pas vide :
                action ← récupérer action_triees[derniere_action]
                si action[cout] + cout_total <= plafond :
                        actions_selectionnees ← actions_selectionnees + action
                        benefice_total ← benefice_total + action[benefice]
                        cout_total ← cout_total + action[cout]
        retourner(actions_selectionnees[benefice], afficher actions_selectionnees)
```

Analyse complexité

Comparaison

Force brute
On parcourt l'ensemble des choix possibles soit 2^n.
Complexité temporelle : **O(2^n)** = 2^20 = 1 048 576 avec 20 le nombre d'éléments à traiter.
Complexité en mémoire : **O(2^n)** = 2^20 = 1 048 576 avec 20 le nombre d'éléments à traiter.

L'obtention de la complexité en mémoire se fait en stockant toutes les combinaisons rassemblées à partir de la complexité temporelle obtenue, pour ensuite les additionner avec chacun des éléments qui ont été nécessaires au calcul de cette complexité. Ainsi, on en fait donc ressortir le nombre de variables dont on a besoin afin d'exécuter notre algorithme. Cependant, la complexité se résumant à la puissance la plus forte, la somme ne sera pas ajoutée pour les calculs de nos complexités en mémoire.

Méthode glouton
Soit *n*, le nombre d'éléments à traiter, éléments qu'il nous faudra ensuite trier, alors :
Complexité temporelle : **O[n*log(n)]** = 20*log(20) = 26,02 avec 20 le nombre d'éléments à traiter.
Complexité en mémoire : **O[n*log(n)]** = 20*log(20) = 26,02 avec 20 le nombre d'éléments à traiter.

Données de Sienna

Préparation des données

On a retenu le fait qu'*il peut y avoir des données manquantes ou incorrectes*.

- On retient donc les actions dont le coût est **nul**
- On retient aussi les actions dont le coût est **négatif**

```
def tri_donnees_sienna():
    for i in range(1, len(liste_actions)):
        if liste_actions[i][1] <= 0:
            liste_actions.pop(i)
            return tri_donnees_sienna()
    print(f"Après le tri des données de Sienna, le nombre d'actions est passé de 1000 à {(len(liste_actions))}.")
tri_donnees_sienna()
```

Résultat du tri :

- **ensemble de données n°1** : **957** actions restantes sur 1001 soit **95,60%**
- **ensemble de données n°2** : **541** actions au lieu de 1000 soit **54,10%**. Ainsi, pour chacun des ensembles de données que nous allons traiter, toutes les valeurs que nous allons manipuler seront strictement positives. Nous pouvons alors traiter les données puis analyser les résultats.

Données de Sienna

Analyse et résultats

Résultats algorithmes
ensemble de données n°1 : **198,50€** de bénéfice pour **499,94€** soit **39,70%.**
ensemble de données n°2 : **197,76€** de bénéfice pour **499,98€** soit **39,55%.**

Sienna récolte quant à elle :
196,61€ pour l'ensemble de données n°1 avec **498,76€** d'investissement soit **39,41%** de bénéfice.
193,78€ pour l'ensemble de données n°2 avec **489,24€** d'investissement soit **39,60%** de bénéfice.

Conclusions :
Les bénéfices obtenus par **les algorithmes** sont d'une moyenne de **39,62%.**
Les bénéfices obtenus par **les choix de Sienna** sont d'une moyenne de **39,50%.**
En plus d'avoir obtenu une **rentabilité** en moyenne supérieure de **0,12%** (algorithme plus efficace),
L'argent récolté est supérieur de **0,96%** (données 1) et **2,02%** (données 2), soit **1,49%** en moyenne.