

AlgoInvest&Trade



Sommaire

- Avantages/inconvénients
- Explication du code
- Comparaison de l'algorithme brute force et optimisé
 - > Big O
 - > Limites de l'algorithme
 - > Performances, efficacité
- Comparaison avec l'algorithme de Sienna
- Conclusion

Algorithme Brute Force

Avantages

- Algorithme de force brute, essayant toutes les combinaisons possibles
- Très bon algorithme si peu de données

Inconvénients

- N'est pas utilisable pour un grand nombre d'actions (complexité : $O(2^n)$)

Test de toutes les combinaisons possibles

Combinaisons avec 3 actions

action1	action2	action3
oui	non	oui
non	non	oui
oui	oui	oui
non	oui	oui
oui	non	non
non	non	non
oui	oui	non
non	oui	non

Toutes les combinaisons
testées avec 20 actions

Nombre d'opérations à tester	
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576

Dépense maximum qui est définie à 500,

```
def bruteforce(depense_max, donnees, lst_actions_selectionnees=[]):  
    if donnees:  
        val1, lst_val1 = bruteforce(depense_max, donnees[1:], lst_actions_selectionnees)  
        action_selection = donnees[0]  
        if action_selection[1] <= depense_max:  
            val2, lst_val2 = bruteforce(depense_max - action_selection[1], donnees[1:],  
                                       lst_actions_selectionnees + [action_selection])  
            if val1 < val2:  
                return val2, lst_val2  
        return val1, lst_val1
```

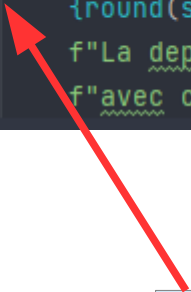
On appelle
récursivement la fonction
Brute force,
Afin de calculer le profit1
et une liste d'action(sans l'action 1)

On récupère la 1ere action
dans une variable de la liste d'action
passé en paramètre de la fonction

On rappelle notre fonction pour calculer le profit2
et une liste d'action.
(sans l'action1)
La dépense max est modifié(sans le prix de l'action)
et on ajoute l'action dans la liste
des actions sélectionnées

On retourne la valeur profit
1 si elle est inférieur a la valeur du profit2
Sinon la valeur du profit2
ainsi que la liste d'actions

```
else:
    return f"la rentabilité maximum obtenue est : \
           {round(sum([i[1] * i[2] for i in lst_actions_selectionnees]), 2)}", \
           f"La dépense maximum est : {sum([i[1] for i in lst_actions_selectionnees])} euros, " \
           f"avec ces actions: {[i[0] for i in lst_actions_selectionnees]}"
```



Si toutes les actions ont été traitées,
alors on renvoie la dépense max
ainsi que la liste des actions sélectionnées

Algorithme optimisé



Avantages

- Efficacité
- Rapidité
- Permet de trouver une solution rapidement avec un grand nombre de données

Inconvénients

- La solution trouvée n'est pas la meilleure

```
start = time.time()

# Récupération des données du CSV
with open('dataset1_Python+P7.csv') as data:
    data = [d for d in csv.DictReader(data, delimiter=',') if float(d['price']) > 0 and float(d['profit']) > 0]

# Filtre par profit
data = sorted(data, key=lambda d: float(d['profit']), reverse=True)
```

On trie nos données pour avoir en tête de liste
les actions avec les meilleurs profits

On récupère les actions de notre CSV
tout en filtrant,
en supprimant toutes données susceptibles d'être négatives

On commence par créer une liste vide
qui contiendra par la suite
toutes nos actions finales


La somme du résultat de nos prix des précédentes actions
Puis on lui ajoute le prix de l'action
et on vérifie que la totalité du montant
des actions ne dépasse pas notre budget maximum défini

```
combination_list = []  
  
for action in data:  
    partial_sum = sumcomb(combination_list) + float(action['price'])  
    if partial_sum <= 500:  
        combination_list.append(action)
```

On boucle sur nos données de notre CSV

Si la somme des résultats est inférieure
au porte-feuille défini
alors on ajoute cette action à notre liste
Si l'on dépasse notre budget, on ne l'ajoute pas
puis on teste la prochaine action

On récupère le prix des actions précédentes
afin de pouvoir l'ajouter à notre nouvelle action



```
def sumcomb(comb):  
    total_price = 0  
    for element in comb:  
        total_price += float(element["price"])  
    return total_price
```

On affiche le profit fait avec toutes nos actions
en mettant en paramètre à notre fonction
toutes nos actions sélectionnées

```
print(f'Le profit des actions est égal à {calculer_profit(combination_list)} €')  
print(f'Le prix total des actions acheté est égal à {sumcomb(combination_list)} € .')
```

```
def sumcomb(comb):  
    total_price = 0  
    for element in comb:  
        total_price += float(element["price"])  
    return total_price  
  
def calculer_profit(comb):  
    total_profit = 0  
    for element in comb:  
        total_profit += (float(element['profit']) * float(element['price'])) / 100  
    return total_profit
```

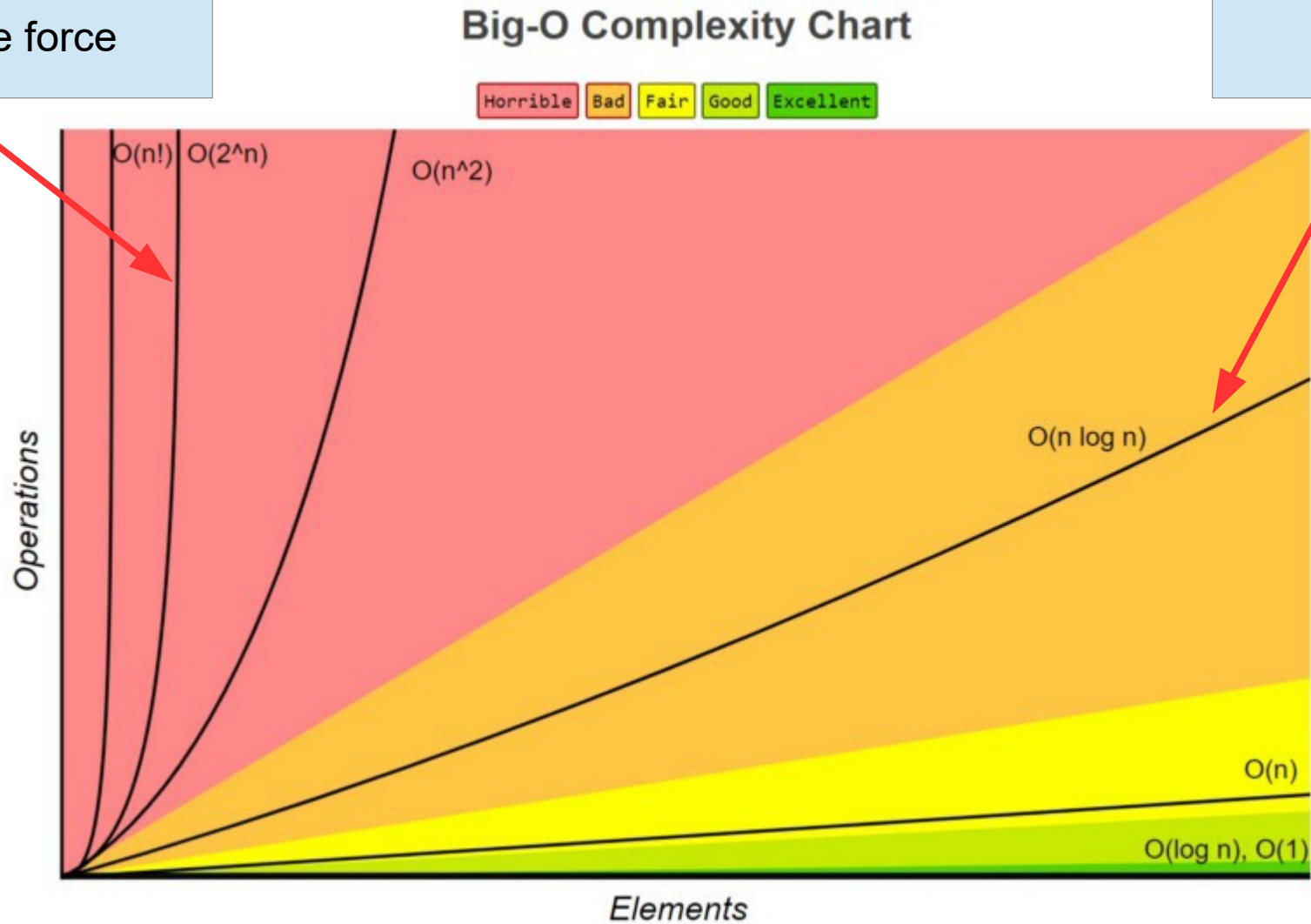
Variable qui contiendra le profit total
de nos actions

On calcule le profit total en faisant une boucle sur notre liste
des actions puis
on multiplie le profit et le prix de l'action
en question et on le divise par 100

Big-O :

Brute force

Optimisé



Les limites de l'algorithme optimisé

- En ne testant pas toutes les combinaisons possibles.
L'algorithme laisse une marge d'erreur sur les bénéfices.

Comparaison :

- **20 actions :**

Brute force :

Optimisé :

- | | | |
|----------------------|------------|----------------|
| • Durée : | 4 secondes | 0,005 secondes |
| • Nombre d'actions : | 10 actions | 14 actions |
| • Bénéfices totaux : | 99,08 | 97,48 |
| • Dépenses max : | 498 | 498 |

Comparaison :

- **1000 actions(dataset1 de Sienna) :**

Optimisé :

Sienna :

- | | | |
|--------------------|--------|--------|
| • Durée : | 0,0084 | |
| • Bénéfices en % : | 39,70 | |
| • Bénéfices : | 198,50 | 196,61 |
| • Dépenses max : | 499,94 | 498,76 |

Comparaison :

- **1000 actions(dataset2 de Sienna) :**

Optimisé :

Sienna :

- | | | |
|--------------------|--------|--------|
| • Durée : | 0,0153 | |
| • Bénéfices en % : | 39,55 | |
| • Bénéfices : | 197,76 | 193,78 |
| • Dépenses max : | 499,98 | 489,24 |

Conclusion

- Brute force :
 - Très efficace sur peu de données. La solution trouvée sera la meilleure.
- Optimisé :
 - Efficace et rapide sur beaucoup de données mais la solution trouvée ne sera pas la meilleure.