

AlgoInvest&Trade



Développement d'algorithmes
permettant de maximiser des
profits



bruteforce.py

Structure du code : bruteforce.py

1) On souhaite tout d'abord obtenir la liste de combinaisons possibles pour un ensemble d'actions données.

Ceci est possible grâce à la fonction `create_set` qui renvoie une liste des combinaisons :
Exemples avec 2 et 3 actions :

➤ `number = 2`



`create_set(number)`



➤ `list_of_set :`
`[[00], [01], [10], [11]]`

➤ `number = 3`



`create_set(number)`



➤ `list_of_set :`
`[[000], [001], [010],
[011], [100], [101],
[110], [111]]`

Il faut donc créer :
 $2^n + 1$ éléments $\Rightarrow O(2^n)$

Il faut garder en mémoire une liste
contenant :
 $2^n + 1$ éléments

`bruteforce_dataset.csv` contient 20 actions

$n = 20$
Combinaisons $= 2^{20} + 1$
 $= 1.048.577$

Structure du code : bruteforce.py

3) On souhaite connaître le coût d'investissement et le bénéfice pour chaque combinaison. On fait appel à la fonction `get_set_of_max_value(list_of_set)` qui parcourt chacune des combinaisons et garde en mémoire le meilleur set (meilleur bénéfice pour un coût maximum donné).

4) Pour chacune de ces combinaisons, la fonction `get_cost_and_value(set)` calcule le coût d'investissement ainsi que le bénéfice.

Exemple pour deux actions :

`get_set_of_max_value(list_of_set) :`

[[00]

[01]

[10]

[11]]



`get_cost_and_value(set)`

Garde en mémoire le meilleur set

Structure du code : bruteforce.py

Action 1 :

- Coût = 7€
- Bénéfice = 5€

Action 2 :

- Coût = 14€
- Bénéfice = 10€

max_cost = 20€

get_set_of_max_value(list_of_set) :

[[00] [01] [10] [11]]

7x0
+14x0
=0

=14

=7

=21

Coût

get_cost_
and_value(set)

5x0
+10x0
=0

=10

=5

=15

Bénéfice



Meilleur investissement (plus haut bénéfice (10€) pour un coût inférieur à 20€)

optimized-glouton.py



Structure du code : optimized-glouton.py

On souhaite investir dans les actions les plus rentables.

1) On souhaite connaître la rentabilité de chacune des actions. Exemple :

Action 1 :

- Coût = 7€
- Bénéfice = 4€
- %bénéfice = 57,14%

Action 2 :

- Coût = 14€
- Bénéfice = 10€
- %bénéfice = 71,43 %

Action 3 :

- Coût = 5€
- Bénéfice = 3€
- %bénéfice = 60 %

2) On trie ces actions de la plus rentable à la moins rentable grâce à la fonction `sort_actions_by_profit_percentage(actions)` qui utilise la fonction `sorted`.

```
sort_actions_by_profit_percentage(actions) : [action 2, action 3, action 1]
```

Il faut créer une liste avec la fonction 'sorted' : $n \times \log(n)$ éléments $\Rightarrow O(N \log N)$

Il faut garder en mémoire une liste contenant : n éléments

Structure du code : optimized-glouton.py

3) On souhaite investir dans les actions les plus rentables dans la mesure où le coût maximum n'est pas dépassé. On fait appel à la fonction `get_invest(sorted_actions)`

Action 1 :	Action 2 :	Action 3 :	
➤ Coût = 7€	➤ Coût = 14€	➤ Coût = 5€	max_cost = 20€
➤ Bénéfice = 4€	➤ Bénéfice = 10€	➤ Bénéfice = 3€	
➤ %bénéfice = 57,14%	➤ %bénéfice = 71,43 %	➤ %bénéfice = 60 %	

`get_invest(sorted_actions) :` Coût = 0 Bénéfice = 0

➤ Etape 1	+ Action 2	14	10
➤ Etape 2	+ Action 3	19	13
➤ Etape 3	+ Action 1	26 > 20	
➤ Etape n	+ Action n

Il faut investir dans l'action 2
Il faut investir dans l'action 3
Il ne faut pas investir dans l'action 1

optimized-dynamic.py



Structure du code : optimized-dynamic.py

1) On construit une liste composée de Nactions + 1 sous-listes de longueur maxcost+1 remplies par des 0. Cette liste peut être représentée sous forme de tableau :

	Action	0	1	1 et 2	1, 2 et 3
	Coût max				
Action 1 Coût = 3 Bénéfice = 2	0	0	0	0	0
	1	0	0	0	0
	2	0	0	0	0
Action 2 Coût = 2 Bénéfice = 1	3	0	0	0	0
	4	0	0	0	0
	5	0	0	0	0
Action 3 Coût = 5 Bénéfice = 4	6	0	0	0	0
	7	0	0	0	0
	8	0	0	0	0
	9	0	0	0	0
	10	0	0	0	0

Il faut garder en mémoire une liste contenant :
 $n \times (\text{maxcost} + 1)$
éléments
 $\Rightarrow O(N)$

On va parcourir chaque élément de cette liste pour en modifier/calculer la valeur

Structure du code : optimized-dynamic.py

2) On va parcourir chaque « colonne » et chaque « ligne » au sein de celle-ci. Pour chaque coût max, on vérifie si la dernière action peut être ajoutée ou si elle doit être exclue.

Action 1
Coût = 3
Bénéfice = 2

Action 2
Coût = 2
Bénéfice = 1

Action 3
Coût = 5
Bénéfice = 4

Action		0	1	1 et 2	1, 2 et 3
Coût max					
0		0	0	0	0
1		0	0	0	0
2		0	0	1	1
3		0	2	2	2
4		0	2	2	2
5		0	2	3	4
6		0	2	3	4
7		0	2	3	5
8		0	2	3	6
9		0	2	3	6
10		0	2	3	7

Il faut garder en mémoire le bénéfice pour chaque cas

Structure du code : optimized-dynamic.py

Action 1
Coût = 3
Bénéfice = 2

Action 2
Coût = 2
Bénéfice = 1

Action 3
Coût = 5
Bénéfice = 4

Action		0	1	1 et 2	1, 2 et 3
Coût max					
0		0	0	0	0
1		0	0	0	0
2		0	0	1	1
3		0	2	2	2
4		0	2	2	2
5		0	2	2 + 1 = 3	4
6		0	2	3	4
7		0	2	3	5
8		0	2	3	6
9		0	2	3	6
10		0	2	3	7

Mathématiquement pour calculer chaque cellule :

1) On vérifie si le coût de l'action est supérieur au coût que l'on est en train de tester :

Exemples :

Le coût de l'action 3 est 5€, ce qui est supérieur au coût testé (2€), il faut **exclure l'action 3**. Ceci correspond à la colonne précédente pour le même coût.

Le coût de l'action 2 est 2€, ce qui est inférieur au coût testé (3€), on peut **envisager d'inclure l'action 2**.

Le coût de l'action 2 est 2€, ce qui est inférieur au coût testé (5€), on peut **envisager d'inclure l'action 2**.

Structure du code : optimized-dynamic.py

Action 1
Coût = 3
Bénéfice = 2

Action 2
Coût = 2
Bénéfice = 1

Action 3
Coût = 5
Bénéfice = 4

Action		0	1	1 et 2	1, 2 et 3
Coût max					
0		0	0	0	0
1		0	0	0	0
2		0	0	1	1
3		0	2	2	2
4		0	2	2	2
5		0	2	3	4
6		0	2	3	4
7		0	2	3	5
8		0	2	3	6
9		0	2	3	6
10		0	2	3	7

Mathématiquement pour calculer chaque cellule :

2) On calcule si il est plus rentable d'inclure ou d'exclure l'action que l'on est en train de tester. L'exclusion correspond à la colonne de gauche.

L'inclusion correspond au bénéfice de l'action testée plus le bénéfice obtenu sans cette action (dans la mesure où le coût max le permet)

Exemples :

Exclusion = 2€
Inclusion = 0 + 1€

MAX=2€

Exclusion = 2€
Inclusion = 2 + 1€

MAX=3€

3) Le résultat final correspond au coût maximum pour toutes les actions.

Structure du code : optimized-dynamic.py

	Action	0	1	1 et 2	1, 2 et 3
	Coût max				
Action 1 Coût = 3 Bénéfice = 2	0				
	1				
	2			Action 2	Action 2
	3		Action 1	Action 1	Action 1
Action 2 Coût = 2 Bénéfice = 1	4		Action 1	Action 1	Action 1
	5		Action 1	Action 1 et 2	Action 3
	6		Action 1	Action 1 et 2	Action 3
	7		Action 1	Action 1 et 2	Action 2 et 3
Action 3 Coût = 5 Bénéfice = 4	8		Action 1	Action 1 et 2	Action 1 et 3
	9		Action 1	Action 1 et 2	Action 1 et 3
	10		Action 1	Action 1 et 2	Action 1 et 2 et 3

Un tableau de taille similaire est créé et parcouru pour garder en mémoire quelles actions sont retenues.

$2 \times n \times (\text{maxcost} + 1)$ éléments



backtesting.py

Exploration de données

	Sienna		optimized-glouton		optimized-dynamic
dataset1	Total cost = 498,76€ Profit = 196,61€	<	Total cost = 499,93€ Profit = 198,49€	<	Total cost = 499,94€ Profit = 198,53€
dataset2	Total cost = 489,24€ Profit = 193,78€	<	Total cost = 499,7€ Profit = 197,72€	<	Total cost = 499,90€ Profit = 197,95€

Conclusion

- L'algorithme optimized-glouton :
 - Mémoire : une liste de n éléments, $O(N)$
 - Efficacité : $O(N \log N)$
- L'algorithme optimized-dynamic :
 - Mémoire : deux listes de $n \times \text{Maxcost}$ éléments, $O(N \times \text{Maxcost})$
 - Efficacité : $O(N \times \text{Maxcost})$
- Exploration des données avec l'algorithme optimized-dynamic :
 - Les profits sont maximisés par rapport aux résultats antérieurs.