



Algo Invest & Trade



P07 - Emile MIATH

SOMMAIRE



1. Graphique de notation Big O

2. Présentation de l'algorithme de force brute

- A. Présentation de la méthode combinations
- B. Présentation des étapes du programme
- C. Avantages et inconvénients
- D. Consommation de la mémoire RAM

3. Présentation de l'algorithme optimisé

- A. Avantages et inconvénients

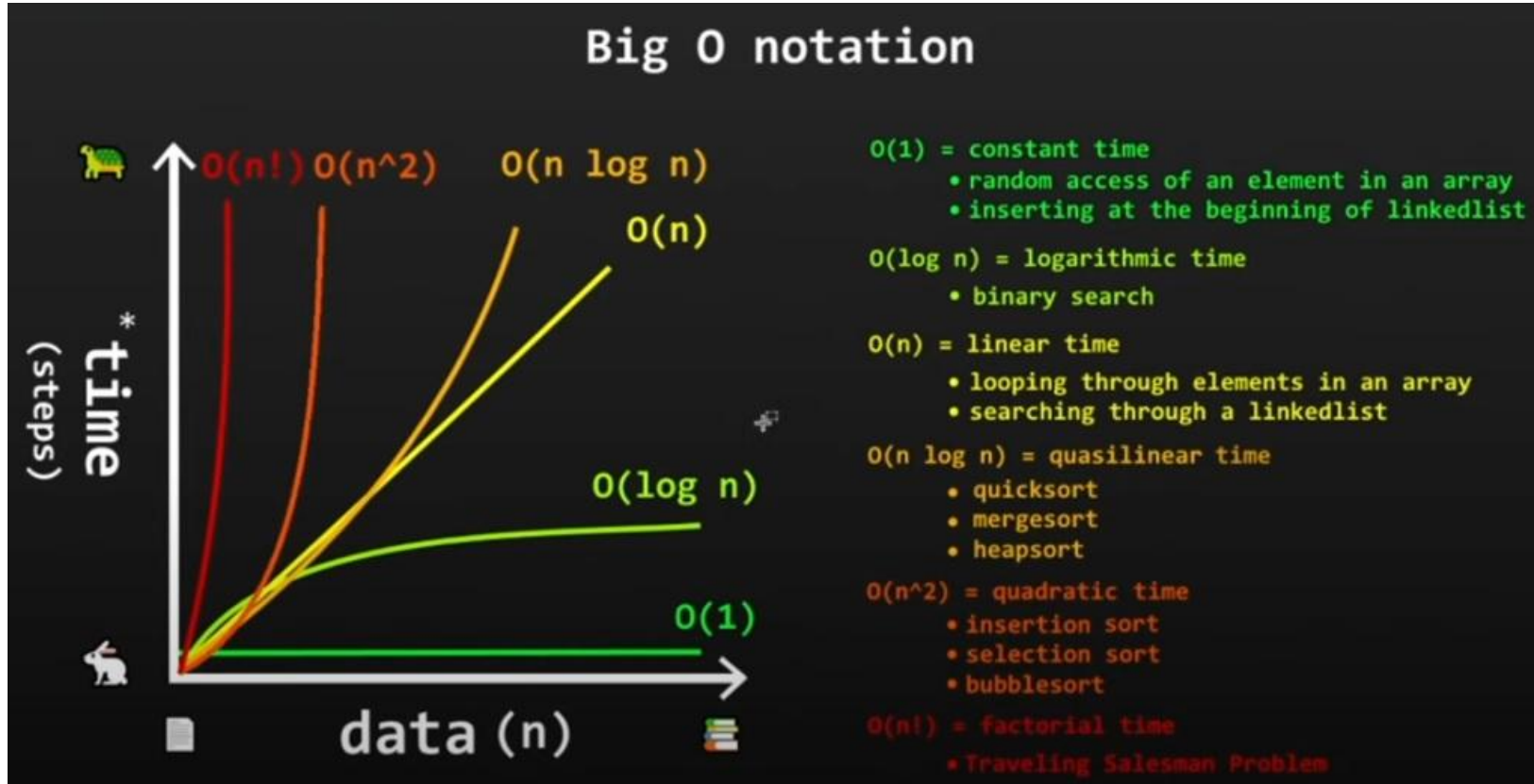
4. Comparaison avec les résultats de Sienna

- A. comparaison Dataset1
- B. comparaison Dataset2

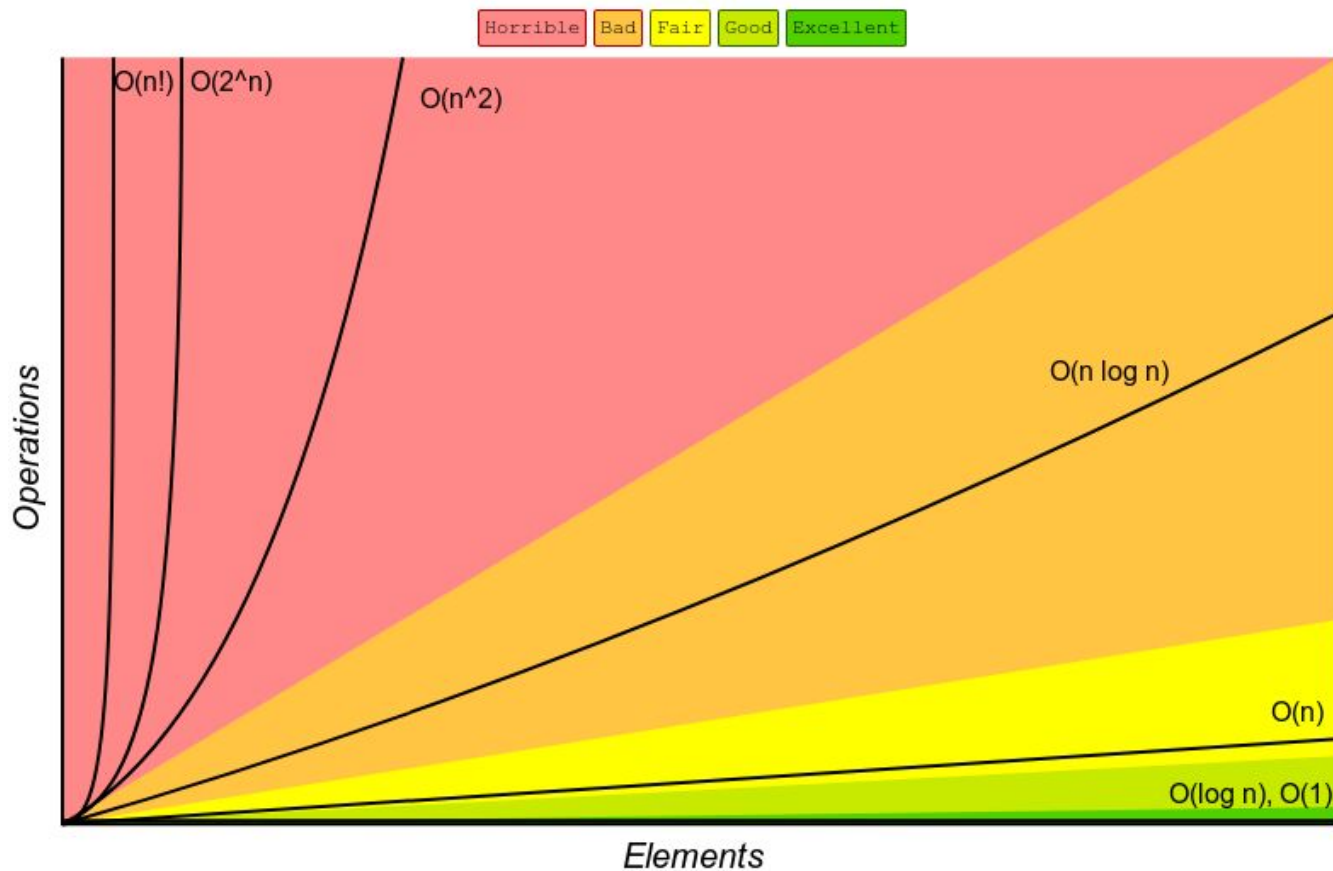
5. Conclusions

1. Les différentes notations Big O


Ce graphique nous montre les différentes courbes de complexités temporelle pour un algorithme donnée défini par leur notation Big O



Cet autre graphique nous montre que plus l'algorithme est pensé sous la forme 2^n tout comme l'algorithme de force brute plus le temps de traitement sera long. Lors du traitement d'un plus grand nombre d'éléments il faudra préférer un algorithme optimisé de type $O(n)$



2A. Présentation de la méthode combinations avec de l'algorithme de force brute



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

Nombre de calculs effectués

On peut voir ici que si la colonne de gauche correspond à un nombre d'actions ou d'éléments, la colonne de droite représente ici le nombre de calcul de combinaison possible.

Ainsi, nous pouvons observer qu'en utilisant un algorithme de type 2^n pour 20 actions, notre ordinateur devrait effectuer 1048576 calculs. A partir de cette observation nous pouvons nous rendre compte qu'en dépit de sa précision cette algorithme sera extrêmement lent pour plus de 20 actions et grignotera drastiquement la mémoire de notre pc.



Afin d'élaborer cet algorithme de force brute j'ai utilisé le module de combinaisons contenue dans la bibliothèque native python : Itertools.

Ce module permet notamment de récupérer toute les actions du dataset et tester toutes les combinaisons possibles les unes avec les autres jusqu'à atteindre la valeur maximale du wallet.

Il va créer des combinaisons sous forme de tuples qui seront implémentés dans une liste.

Ainsi, si nous n'avions que 4 actions à combiner, la première combinaison sera

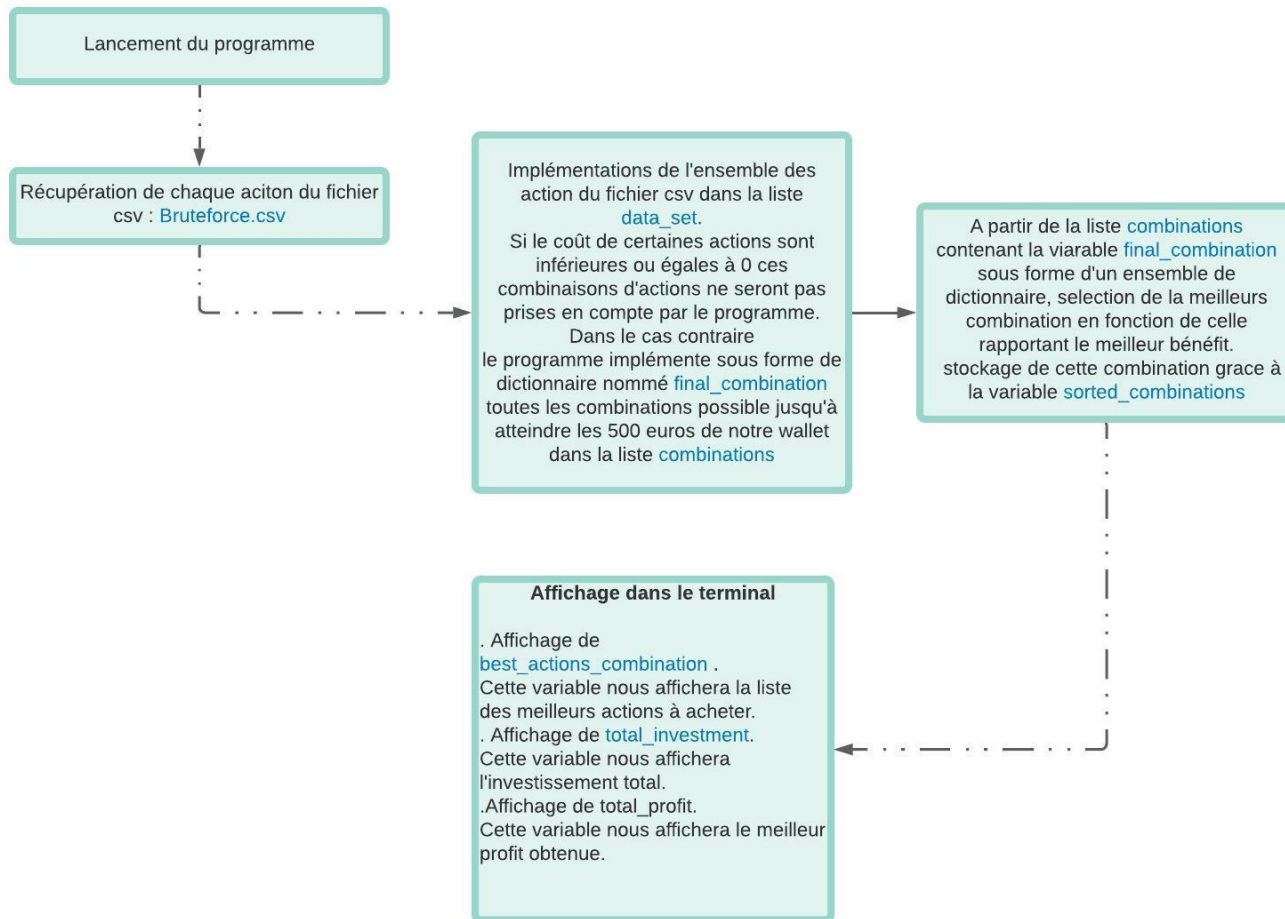
[(action-1,20,5), (action-2,30,10)] et la dernière

[(action-1,20,5), (action-2,30,10), (action-3,50,15), (action-4,70,20)]

il sera nécessaire de faire 16 calculs pour déterminer chaque combinaisons.

A chaque ajout d'une nouvelle action nous aurons alors un calcul de l'ordre de calculs (calculs^2)

2B. Étapes d'exécution du programme



2C. Avantages et inconvénients de l'algorithme de force brute



Avantages:

- . Taux d'efficacité s'approchant des 100%
- . Marge d'erreur extrêmement réduite
- . Méthode parfaite du point de vue résultats pour un petit nombre d'éléments à traiter

Inconvénients:

- . Complexité exponentielle
- . Énormément Chronophage et consomme énormément de mémoire

2D. Algorithme force brute: consommation de la mémoire RAM

Line #	Mem usage	Increment	Occurrences	Line Contents
26	19.4 MiB	19.4 MiB	1	@profile
27				def brute_force_algorithme(money_wallet):
28				
29	19.4 MiB	0.0 MiB	1	data_set = []
30	19.4 MiB	0.0 MiB	1	combinations = []
31				
32	19.4 MiB	0.0 MiB	1	primary_csv_file = './csv_f/Bruteforce.csv'
33				
34	19.4 MiB	0.0 MiB	1	with open(primary_csv_file, newline='') as csv_file:
35	19.4 MiB	0.0 MiB	1	reader = csv.DictReader(csv_file)
36	19.4 MiB	0.0 MiB	21	for element in reader:
37	19.4 MiB	0.0 MiB	20	action_productivity = float(element["cost"]) * float(element["benefit"]) / 100
38	19.4 MiB	0.0 MiB	20	data_set.append(Dataset(element["name"], (element["cost"]), action_productivity).serialize
				d_dataset())
39				
40	572.0 MiB	-202.4 MiB	21	for i in range(1, len(data_set) + 1):
41	572.0 MiB	-28920382.1 MiB	1048595	for combination in cbts(data_set, i):
42	572.0 MiB	-28920046.6 MiB	1048575	final_combination = {
43	572.0 MiB	-28920281.7 MiB	1048575	"action_list": [],
44	572.0 MiB	-28920289.0 MiB	1048575	"total_cost": 0,
45	572.0 MiB	-28920293.4 MiB	1048575	"total_benefit": 0,
46				}
47	572.0 MiB	-351878499.5 MiB	11534335	for element in combination:
48				
49	572.0 MiB	-322958395.0 MiB	10485760	if (final_combination["total_cost"] + element[1]) <= money_wallet:
50	572.0 MiB	-314521684.6 MiB	10149463	final_combination["action_list"].append(element)
51	572.0 MiB	-314521820.5 MiB	10149463	final_combination["total_cost"] += element[1]
52	572.0 MiB	-314521905.2 MiB	10149463	final_combination["total_benefit"] += element[2]
53	572.0 MiB	-314521864.8 MiB	10149463	combinations.append(final_combination)
54	572.0 MiB	-314522024.6 MiB	10149463	print
55				
56	572.0 MiB	-1407299142.1 MiB	20298927	sorted_combinations = max(combinations, key=lambda combination: combination["total_benefit"])
57				
58	489.8 MiB	-82.2 MiB	1	best_actions_combination = sorted_combinations["action_list"]
59	489.8 MiB	0.0 MiB	1	total_investment = sorted_combinations["total_cost"]
60	489.8 MiB	0.0 MiB	1	total_profit = sorted_combinations["total_benefit"]
61				
62				# Reports
65	490.0 MiB	0.0 MiB	11	for element in best_actions_combination:
66	490.0 MiB	0.0 MiB	10	print(element[0])
67	490.0 MiB	0.0 MiB	1	print("_____")
68	490.1 MiB	0.0 MiB	1	print(f"\nFor a total investment of {total_investment}€ \n")

3. Algorithme Optimisé



L'algorithme optimisé est basé une solution d'algorithme dynamique.

Dans un tableau à deux dimensions contenant 0 dans chaque lignes et chaque colonnes .

Le nombre de ligne correspond au nombre d'actions présentes dans le dataset tandis que le nombre de colonnes correspond à chaque unité du budget maximum investi.

Il parcourt alors les cellules de cette matrice une par une, une seule et unique fois.

Son exploitation permet alors de déterminer la meilleur liste d'actions à acheter.

3A. Avantages et inconvénients de l'algorithme optimisé



Avantages:

- . Exécutions très rapide en moins d'une seconde.
- . Tri effectué selon plusieurs paramètres et comparaison pour une sortie optimale

Inconvénients:

- . Marge d'erreur potentielle par rapport à la méthode de force brute

4A. Algorithme optimisé : Comparaison des résultats avec Sienna

Dataset 1

For the better investment you need to buy this list of actions :

Share-DBUJ
Share-KMTG
Share-GHIZ
Share-CYYC
Share-NHWA
Share-UEZB
Share-IQMC
Share-LPDM
Share-MTLR
Share-GTQK
Share-FKJW
Share-EVUW
Share-QLMK
Share-CGJM
Share-WPLI
Share-LGWG
Share-ZSDE
Share-LOKP
Share-SKKC
Share-STKT
Share-QQTU
Share-PBXL
Share-KGQI
Share-CBNY
Share-XJMO
Share-LRBZ
Share-EMOV
Share-IFCP

For a total investment of 500€

You will win : 199.93€

That means that your return on investment is : 39.99 %

Program executed in 0.41 seconds

Sienna bought:

Share-GRUT

Total cost: 498.76€

Total return: 196.61€

4A. Algorithme optimisé : Consommation de la mémoire RAM, Dataset1

Line #	Mem usage	Increment	Occurrences	Line Contents
19	19.9 MiB	19.9 MiB	1	@profile
20				def optimized_algorithm(wallet_cost, actions):
21	19.9 MiB	0.0 MiB	1	number_of_actions = len(actions)
22	23.9 MiB	4.0 MiB	482331	matrix = [[0 for x in range(wallet_cost + 1)] for x in range(len(actions) + 1)]
23	24.2 MiB	0.3 MiB	1	print(matrix)
24	25.2 MiB	-3.4 MiB	957	for i in range(1, len(actions) + 1):
25	25.2 MiB	-1684.6 MiB	478956	for w in range(1, wallet_cost + 1):
26	25.2 MiB	-1681.2 MiB	478000	if actions[i-1][1] <= w:
27	25.2 MiB	-3194.6 MiB	909100	matrix[i][w] = max(actions[i-1][2] + matrix[i-1][w-actions[i-1][1]],
28	25.2 MiB	-1596.3 MiB	454550	matrix[i-1][w])
29				else:
30	25.2 MiB	-83.9 MiB	23450	matrix[i][w] = matrix[i-1][w]
31				
32	25.2 MiB	0.0 MiB	1	w = wallet_cost
33	25.2 MiB	0.0 MiB	1	n = number_of_actions
34	25.2 MiB	0.0 MiB	1	final_combination = []
35				
36	25.2 MiB	0.0 MiB	958	while w >= 0 and n >= 0:
37	25.2 MiB	0.0 MiB	957	if matrix[n][w] != matrix[n-1][w]:
38	25.2 MiB	0.0 MiB	28	final_combination.append(actions[n-1])
39	25.2 MiB	0.0 MiB	28	w -= actions[n-1][1]
40	25.2 MiB	0.0 MiB	957	n -= 1
41				
42				
43	25.2 MiB	0.0 MiB	31	actions_names = [x[0] for x in final_combination]
44	25.2 MiB	0.0 MiB	31	total_investment = sum([(x[1]) for x in final_combination])
45	25.2 MiB	0.0 MiB	31	total_profit = sum([(x[2]) for x in final_combination])
46				
47	25.2 MiB	0.0 MiB	1	print("\n For the better investment you need to buy this list of actions : \n")
48	25.2 MiB	0.0 MiB	1	print("----- \n")
49	25.2 MiB	0.0 MiB	29	for actions_names in final_combination:
50	25.2 MiB	0.0 MiB	28	print(actions_names[0])
51	25.2 MiB	0.0 MiB	1	print("\n ----- \n")
52	25.2 MiB	0.0 MiB	1	print(f"")
53	25.2 MiB	0.0 MiB	1	print(f"\n For a total investment of {total_investment}€")
54	25.2 MiB	0.0 MiB	1	print(f"\n You will win : {round(total_profit, 2)}€")
55	25.2 MiB	0.0 MiB	2	print(
56	25.2 MiB	0.0 MiB	1	f"\n That means that your return on investment is : {round(total_profit / total_investment * 100,
				2)} % \n")

4B. Algorithme optimisé : Comparaison des résultats avec Sienna

Dataset 2

For the better investment you need to buy this list of actions :

Share-IXCI
Share-FWBE
Share-ZOFA
Share-PLLK
Share-MEQV
Share-LXZU
Share-PATS
Share-SCWM
Share-ZLMC
Share-VCXT
Share-NDKR
Share-ALIY
Share-JWGF
Share-FCHD
Share-LKSD
Share-JGTW
Share-VCAX
Share-LFXB
Share-DWSK
Share-UPCV
Share-DYVD
Share-XQII
Share-OAVO
Share-ROOM
Share-YCGH

For a total investment of 500€

You will win : 199.04€

That means that your return on investment is : 39.81 %

Program executed in 0.21 seconds

Sienna bought:

Share-ECAQ 3166
Share-IXCI 2632
Share-FWBE 1830
Share-ZOFA 2532
Share-PLLK 1994
Share-YFVZ 2255
Share-ANFX 3854
Share-PATS 2770
Share-NDKR 3306
Share-ALIY 2908
Share-JWGF 4869
Share-JGTW 3529
Share-FAPS 3257
Share-VCAX 2742
Share-LFXB 1483
Share-DWSK 2949
Share-XQII 1342
Share-ROOM 1506

Total cost: 489.24€

Profit: 193.78€

4B. Algorithme optimisé : Consommation de la mémoire RAM, Dataset2

Line #	Mem usage	Increment	Occurrences	Line Contents
16	19.8 MiB	19.8 MiB	1	@profile
17				def optimized_algorithm(wallet_cost, actions):
18	19.8 MiB	0.0 MiB	1	number_of_actions = len(actions)
19	22.1 MiB	2.3 MiB	273171	matrix = [[0 for x in range(wallet_cost + 1)] for x in range(len(actions) + 1)]
20				
21	22.9 MiB	-4.0 MiB	542	for i in range(1, len(actions) + 1):
22	22.9 MiB	-2039.8 MiB	271041	for w in range(1, wallet_cost + 1):
23	22.9 MiB	-2035.7 MiB	270500	if actions[i-1][1] <= w:
24	22.9 MiB	-3880.1 MiB	514398	matrix[i][w] = max(actions[i-1][2] + matrix[i-1][w-actions[i-1][1]],
25	22.9 MiB	-1939.3 MiB	257199	matrix[i-1][w])
26				else:
27	22.9 MiB	-95.7 MiB	13301	matrix[i][w] = matrix[i-1][w]
28				
29	22.9 MiB	0.0 MiB	1	w = wallet_cost
30	22.9 MiB	0.0 MiB	1	n = number_of_actions
31	22.9 MiB	0.0 MiB	1	final_combination = []
32				
33	22.9 MiB	0.0 MiB	543	while w >= 0 and n >= 0:
34	22.9 MiB	0.0 MiB	542	if matrix[n][w] != matrix[n-1][w]:
35	22.9 MiB	0.0 MiB	25	final_combination.append(actions[n-1])
36	22.9 MiB	0.0 MiB	25	w -= actions[n-1][1]
37	22.9 MiB	0.0 MiB	542	n -= 1
38				
39				
40	22.9 MiB	0.0 MiB	28	actions_names = [x[0] for x in final_combination]
41	22.9 MiB	0.0 MiB	28	total_investment = sum([(x[1]) for x in final_combination])
42	22.9 MiB	0.0 MiB	28	total_profit = sum([(x[2]) for x in final_combination])
43				
44	22.9 MiB	0.0 MiB	1	print("\n For the better investment you need to buy this list of actions : \n")
45	22.9 MiB	0.0 MiB	1	print("----- \n")
46	22.9 MiB	0.0 MiB	26	for actions_names in final_combination:
47	22.9 MiB	0.0 MiB	25	print(actions_names[0])
48	22.9 MiB	0.0 MiB	1	print("\n ----- \n")
49	22.9 MiB	0.0 MiB	1	print(f"")
50	22.9 MiB	0.0 MiB	1	print(f"\n For a total investment of {total_investment}€")
51	22.9 MiB	0.0 MiB	1	print(f"\n You will win : {round(total_profit, 2)}€")
52	22.9 MiB	0.0 MiB	2	print(
53	22.9 MiB	0.0 MiB	1	f"\n That means that your return on investment is : {round(total_profit / total_investment * 100,
				2)} % \n")

5. Conclusion



Algorithme brute force

VS

Algorithme Optimisé

**Pour n = nombre d'
éléments à analyser**

Brute Force

- . Complexité de temps $O(2^n)$
- . Complexité de mémoire $O(2^n)$
- . Exponentielle

Optimisé

- . Complexité de temps $O(n)$
- . Complexité de mémoire $O(n)$
- . linéaire