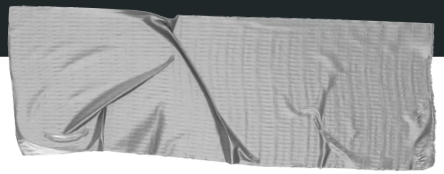
Résolvez des problèmes en utilisant des algorithmes en Python

LEGRAND Philippe

But Poursuivi:

- Utiliser divers jeux de données afin de créer des algorithmes pour résoudre un problème.
- Comprendre les enjeux des algorithmes.
- Comparer les différents résultats obtenus.
- Déterminer les avantages et inconvénients des choix effectués.



1. Définition:

Comme indiqué dans le titre du projet, l'algorithme est:

- **Un ensemble de règles, d'opérations,**
- **ayant pour but de résoudre un problème donné.**

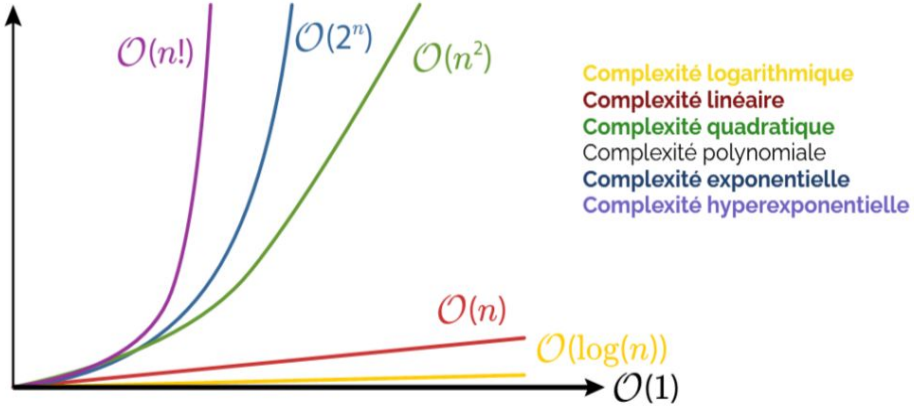
- **La performance d'un algorithme est appelée complexité, elle se mesure de deux manières:**
  - *Temporelle*
  - *Spatiale*

La complexité d’un algorithme:

C’est en quelque sorte sa performance, elle se mesure de deux manières, temporelle et spatiale.

1. **La complexité Temporelle:**

   C’est une approximation du nombres d’opérations, sa vitesse d'exécution.



2. **La complexité Spatiale:**

   C’est une approximation de la mémoire occupée pendant l’exécution des opérations.

AlgoInvest&Trade

Le sujet traité:

- Une société financière, doit fournir à ses clients, pour un montant d'investissement donné, le meilleur choix d'actions, en fonction de leur rentabilité en % sur deux ans.

- Un Algorithme de Brute-Force est demandé
- Puis un algorithme optimisé.

L’algorithme de Brute-Force:

- Une solution qui vise à tester toutes les possibilités d’un problème.

- Le choix d’une action étant possible ou non, la construction d’un algorithme sur la base du système binaire a été implémentée.

- Une fois le nombre de possibilités calculé, seuls les choix possibles sont retenus pour évaluer les possibilités.

Calcul des combinaisons:

rang	Liste Actions Position binaire inversée [5, 7, 9]	Liste Actions Position décimal [5, 7, 9]	Liste Actions Index [5, 7, 9]
0	0		
1	1	[5]	Action[0]
2	0 1	[7]	Action[1]
3	1 1	[5, 7]	Action[0] Action[1]
4	0 0 1	[9]	Action[2]
5	1 0 1	[5, 9]	Action[0] Action[2]
6	0 1 1	[7, 9]	Action[1] Action[2]
7	1 1 1	[5, 7, 9]	Action[0] Action[1] Action[2]

Pour 3 actions, le nombre de choix possibles est de 2^3 soit 8

1. On calcule le nombre de possibilités `n_comb`, et on initialise les variables `best cost, best profit, best_action_list`

2. **1ère boucle:** pour chaque possibilité de `n_comb`
    1. On initialise les variables `cost_sum, profit_sum, combination`
    2. On crée un tuple `bin_comb = (index, binaire inversé)`
3. **2ème boucle dans la 1ère:** pour chaque index de notre liste inversée `bin_comb`
    1. **Si** la position `"1"` existe, on incrémente nos variables.

4. Enfin, **dans la 1ère boucle** **Si** le coût cumulé est inférieur à l'investissement maximum **et** le profit cumulé est supérieur au profit précédent.

Alors on initialise nos variables qui seront retournées comme meilleur solution `return best_cost, best_profit, best_action_list`

```
n_comb = 2 ** (len(action_cost_list))
best_cost, best_profit, best_action_list = 0, 0, []



for possibilities in range(1, n_comb):
    cost_sum, profit_sum, combination = 0, 0, []
    bin_comb = list(enumerate(reversed(bin(possibilities)[2:])))
    for index in bin_comb:
        if index[1] == "1":
            cost_sum += int(action_cost_list[index[0]])
            profit_sum += int(action_cost_list[index[0]]) * \
                int(action_profit[index[0]]) / 100
            combination.append(action_name_list[index[0]])



    if cost_sum < max_investment and profit_sum > best_profit:
        best_cost = cost_sum
        best_profit = profit_sum
        best_action_list = combination

return best_cost, best_profit, best_action_list
```

Conclusion sur le Brute-Force: $O(2^N)$

- Pour 20 actions, la solution implémentée prend environ 5 secondes, et l'on compte déjà plus d'1 million de possibilités.

- Si le brute force donne la réponse la plus pertinente à chaque fois, plus le nombre d'actions à traiter est grand, plus les possibilités augmentent. Il s'agit d'une courbe exponentielle : 2^N

- Ainsi avec 100, 200 actions, ou plus, le résultat atteindra rapidement des milliers d'années et cette solution ne sera pas soutenable.

L’algorithme optimisé:

- Pour l’optimisation c’est l'algorithme dit glouton qui a été choisi.

- Cet algorithme suit la logique d’un choix optimal à chaque étape de sa réalisation.

PARTIE 1 (1ère optimisation)

1. On initialise les variables `result = []`, `actionYield = []` `index = -1`

2. **1ère boucle,** pour chaque élément dans l'intervalle du nombre d'action;

    1. On ajoute le bénéfice sur 2 ans en Euros à la variable `actionYield`

    2. Puis on crée un dictionnaire avec l'ensemble des valeurs ; Action, Prix, %, et Bénéfice.

3. **Fin de la 1ère boucle,** on trie notre dictionnaire du bénéfice le plus élevé au bénéfice le moins élevé.
`list_of_data = sorted(result, key=lambda x: -x['Yield'])`

```
result = []
actionYield = []
index = -1


for element in range(len(defAction)):
    index += 1

    actionYield.append(
        round((defPrice[element] * (defPercentage[element] / 100)), 2))


    result.append({'Action': defAction[index],
                   'Price': defPrice[index],
                   'Percentage': defPercentage[index],
                   'Yield': actionYield[index]})


list_of_data = sorted(result, key=lambda x: -x['Yield'])
```

PARTIE 2 (2ème optimisation)

4. On initialise les variables `spent = 0`, `action_to_buy = []` et `benefit = 0`

5. **2ème boucle**, pour chaque élément, dans l'intervalle du nombre d'action;

    1. **Si** le prix est supérieur à 0.0 **et** le total dépensé + le prix est <= au maximum à investir:
    On incrémente le prix, le bénéfice et on ajoute l'action à la liste d'achat.

6. Enfin, on retourne les résultats obtenus.

```
spent = 0
action_to_buy = []
benefit = 0


for element in range(len(list_of_data)):


    if list_of_data[element]["Price"] > 0.0 and spent + \
        list_of_data[element]["Price"] <= max_investment:
        spent += list_of_data[element]["Price"]
        benefit += list_of_data[element]["Yield"]
        action_to_buy.append(list_of_data[element]["Action"])



return spent, action_to_buy, round(benefit, 2)
```

Conclusion notre algorithme Optimisé: O(n)

- L’algorithme glouton, permet grâce à son optimisation par étape, un temps d’exécution très performant, moins d’une seconde. Il est dit linéaire.

- Il n’en demeure pas moins, qu’il ne permet aucun retour en arrière une fois qu’un choix est fait. Lorsqu’une donnée est traitée, elle sort du champ des possibles, et ne peut pas être exploitée dans un second temps, pour affiner le résultat.

- S’il gagne en temps d’exécution, il perd dans la finesse de résultat obtenu.

Comparaison des solutions optimisées:

Résultat du premier jeu de données : Dataset1							
SIENNA				PHILIPPE			
Action	Prix	%/2ans	Bénéfices	Action	Prix	%/2ans	Bénéfices
Share-GRUT	498.76	39.42	196.61€	Share-GRUT	498.76	39.42	196,61 €
				Share-CBNY	1.22	39.31	0,48 €
				Share-MLGM	0.01	18.86	0,02 €
Total	**498,76 €**	**196,61 €**		**Total**	**499,99 €**	**197,13 €**	

CONCLUSION :

Dans ce dataset, les bénéfices sont plus importants chez Philippe qui a 3 actions achetées. Sienna n'a acheté qu'une action et n'a pas atteint le montant maximal à investir. Néanmoins on voit que la différence n'est pas significative, la question de l'investissement sur ces 2 actions supplémentaires se pose. On peut supposer qu'une condition de rendement minimal est présente dans l'algorithme de Sienna ce qui conduit à l'achat d'une seule action.

Résultat du deuxième jeu de données : Dataset2							
SIENNA				PHILIPPE			
Action	Prix / €	%/2ans	Bénéfices	Action	Prix / €	%/2ans	Bénéfices
Share-ECAQ	31,66	39,49	12,50	Share-JWGF	48,69	39,93	19,44
Share-IXCI	26,32	39,4	10,37	Share-MBQU	51,46	35,78	18,41
Share-FWBE	18,3	39,82	7,29	Share-QEVK	49,77	34,38	17,11
Share-ZOFA	25,32	39,78	10,07	Share-DLNE	44,06	36,74	16,19
Share-PLLK	19,94	39,91	7,96	Share-IJFT	40,91	38,89	15,91
Share-YFVZ	22,55	39,1	8,82	Share-ANFX	38,54	39,72	15,31
Share-ANFX	38,54	39,72	15,31	Share-MALJ	46,37	32,88	15,25
Share-PATS	27,7	39,97	11,07	Share-OPBR	39	38,95	15,19
Share-NDKR	33,06	39,91	13,19	Share-FWMV	41,68	35,08	14,62
Share-ALIY	29,08	39,93	11,61	Share-HATC	43,45	34,14	14,83
Share-JWGF	48,69	39,93	19,44	Share-XGNC	41,86	35,14	14,71
Share-JGTW	35,29	39,43	13,91	Share-XQII	13,42	39,51	5,30
Share-FAPS	32,57	39,54	12,88	Share-DYVD	0,28	10,25	0,03
Share-VCAX	27,42	38,99	10,69	Share-LKSD	0,12	9,14	0,01
Share-LFXB	14,83	39,79	5,90				
Share-DWSK	29,49	39,35	11,60				
Share-XQII	13,42	39,51	5,30				
Share-ROOM	15,06	39,23	5,91				
TOTAL	489,24	193,83		TOTAL	499,61	182,31	

CONCLUSION :

Sur ce dataset on constate que le résultat est meilleur chez Sienna. Ici encore on voit que l'algorithme de Sienna tend vers un rendement maximum par actions, alors que l'algorithme glouton tend vers une résolution rapide du problème et un budget maximum à ne pas dépasser. Cette structure alimente la thèse d'une condition minimale de rendement pour la choix de l'action. On constate d'ailleurs qu'aucune action en dessous de 38,99 % n'est sélectionnée. Le pré-trie par ordre de rendement de l'algorithme glouton, n'est donc pas suffisant pour obtenir une solution optimisée.

Conclusion:

Si l’élaboration d'algorithmes très complexes n’est pas toujours au cœur du travail de développeur, les principes fondamentaux qui les régissent le sont.

La complexité spatiale et temporelle permettent aisément de comprendre comment ces éléments impactent la soutenabilité et l'efficacité du code.

Il est donc important, d’avoir toujours à l’esprit ce ratio, pour implémenter un code efficace et qui supporte de forte sollicitation.