AlgoInvest&Trade

La société *AlgoInvest&Trade* est une société financière spécialisée dans l'investissement. La société cherche à optimiser ses stratégies d'investissement à l'aide d'algorithmes, afin de dégager davantage de bénéfices pour ses clients.

Plusieurs contraintes sont appliquées pour faire les choix d'actions à acheter :

- Une action ne peut être achetée qu'une seule fois.
- Une action ne peut se vendre fractionnée.
- Le montant total des achats ne peut dépasser 500€.

Plusieurs algorithmes sont proposés pour faire ces choix. Les voici.

Partie 1 : bruteforce

Cette méthode consiste à tester l'ensemble des combinaisons une à une.

Calcul du nombre de possibilités :

- 20 actions comprenant 2 possibilités :
  - Acheter l'action
  - Ne pas acheter l'action

Donc 2^{20} = 1 048 576 possibilités

Proposition d'algorithme :

A partir de la liste *(listActions[2^{20}])* de toutes les combinaisons possibles, sélectionner celles dont le coût ne dépasse pas 500€.

Pour chacune de ces lignes, calculer le rendement et vérifier s'il est supérieur au meilleur rendement précédent.

Si supérieur : stockage de la combinaison d'action dans une liste (*meilleurCombinaisonActions[]*) et stocker le rendement dans une variable (*meilleurRendement*).

Pour faire le choix des actions à acheter, j'utilise un « compteur binaire » sur 20 digits :

1 = 00000000000000000001

2 = 00000000000000000010

3 = 00000000000000000011

Etc.

Chaque *digits* correspond à une action et le chiffre (0 ou 1) à un acte :

- 0 = ne pas acheter l'action
- 1 = acheter l'action

Prenons l'exemple suivant avec le chiffre 405 874 (base 10) = 01100011000101110010 (base 2)

BIN	0	1	1	0	0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	0
N° action	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Cette combinaison représente donc l'achat des actions 19, 18, 14, 13, 9, 7, 6, 5 et 2.

Il faudra donc ensuite calculer le rendement et le comparer au précédent et déterminer quel est le meilleur.

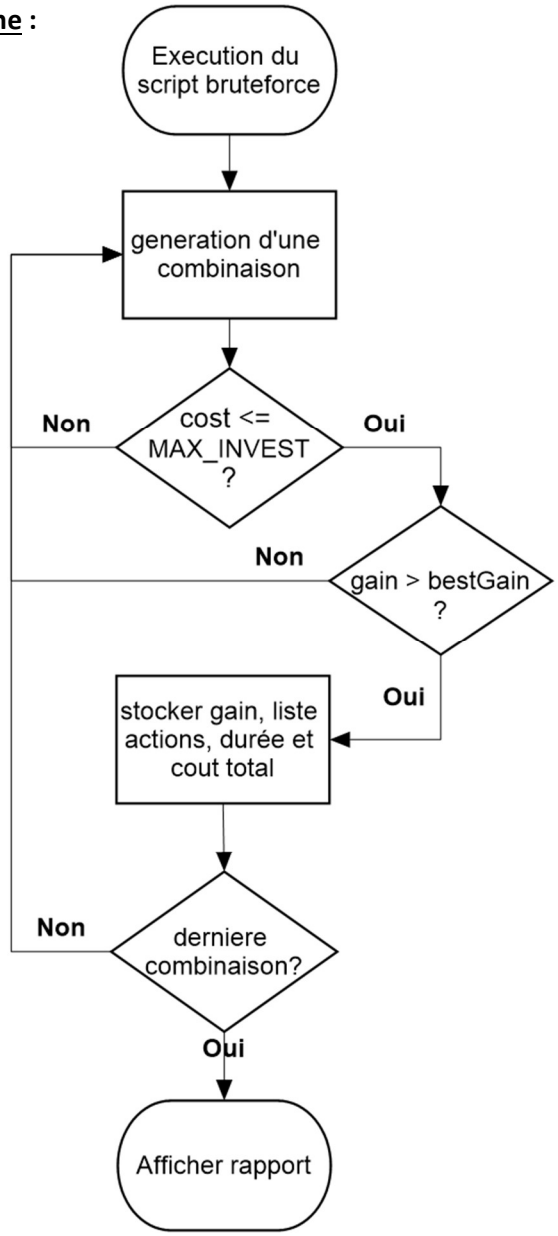
Le calcul de complexité temporelle est $O(2^n)$. Etant donné que pour déduire la meilleure combinaison nous n'avons pas d'autres choix que de toutes les tester, cela rend constant le temps de traitement donc dans notre contexte, nous pouvons en déduire que le calcul de complexité temporelle sera égal à $O(2^{20})$ soit 1 048 576 combinaisons.

La complexité spatiale est 2^n. Comme dit précédemment et dans notre contexte, pas d'autres choix que de tester toutes les combinaisons donc le traitement s'exécutera 2^n fois soit 1 048 576 fois.

Pseudo-code :

```
Algorithme meilleurRendement
Variable
	listCombinationActions : LISTE CONTENANT TOUTES LES COMBINAISONS DE SELECTION D'ACTIONS A ACHETER
	meilleurCombinaisonActions[] : LISTE VIDE POUR STOCKER LA COMBINAISON D'ACTION FINALE A ACHETER
	meilleurRendement <- 0 : RENDEMENT ASSOCIE A LA MEILLEURE COMBINAISON D'ACTION
	MAX_INVEST <- 500 : VARIABLE CONTENANT L'INVESTISSEMENT MAXIMUM A NE PAS DEPASSER
Début
	Pour i <- 0 ; taille listCombinationActions ; i = i + 1
		SI cout(listCombinationActions[i]) =< MAX_INVEST :
			SI rendement de listCombinationActions[i] > meilleurRendement :
				meilleurCombinationActions[] <- listCombinationActions[i]
				meilleurRendement <- rendement de listCombinationActions[i]
			Fin SI
		Fin SI
	Fin POUR
Fin
```

Organigramme :



Depuis dossier *controllers*: `python.exe .\bruteforce.py ..\datas\dataset.csv`

DecToBin(i) génère une combinaison binaire utilisé pour selectionner des actions

```
if GetActionsCost(binary_index, actionsCostsList) <= MAX_INVEST :
```

```
if gain > bestGain:
```

```
bestGain = gain
for action in COMBINATIONS_ACTIONS_LIST:
  actions += action + ", "
actionsCost = GetActionsCost(binary_index, actionsCostsList)
str(datetime.datetime.now() - start
```

fin de la boucle For

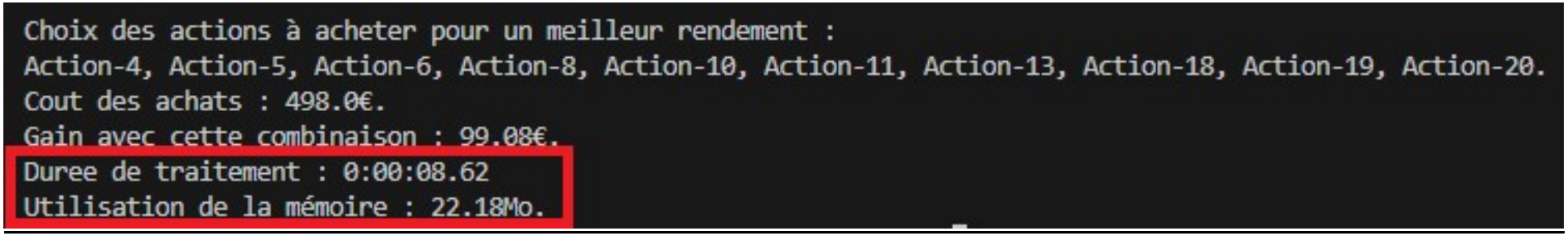
```
bruteforceReport(maxProfit[0], maxProfit[1], maxProfit[2], maxProfit[3])
```

Conclusion :

Avantage : cette méthode teste toutes les combinaisons possibles.

Inconvénient : si on ajoute plus d'éléments (nombre d'actions) cela créé plus de combinaisons et donc le temps de traitement sera allongé. Par exemple, si nous ajoutons cinq actions de plus dans le fichier, la complexité temporelle passera de $O(2^{20})$ à $O(2^{25})$. Le temps de traitement en sera donc lourdement impacté. Le calcul de complexité spatiale sera également impacté pour la même raison.

Dans notre contexte, le temps de traitement reste malgré tout correct (environ 10 secondes) car très peu d'actions à tester. Idem pour l'espace mémoire utilisé, car la boucle ne se répète pas beaucoup.



Partie 2 : Optimisation

Algorithme glouton

Le but de cette méthode est d'acheter le maximum d'actions en sélectionnant en priorité celles qui rapportent le plus tout en ne dépassant pas la limite d'achat maximum de 500€. Les actions qui rapportent le plus sont celles dont le ratio profit et gain est le meilleur. Je prends donc en compte deux critères au lieu d'un ce qui affine le tri. La formule sera donc : PROFIT + (COST * (PROFIT / 100)).

Ci-dessous les listes fournies dans le fichier :

```
Costs = [20, 30, 50, 70, 60, 80, 22, 26, 48, 34, 42, 110, 38, 14, 18, 8, 4, 10, 24, 114]
```

```
Profits = [5, 10, 15, 20, 17, 25, 7, 11, 13, 27, 17, 9, 23, 1, 3, 8, 12, 14, 21, 18]
```

Ci-dessous les listes triées par ordre de rentabilité :

```
Costs = [80, 114, 34, 70, 38, 60, 24, 42, 50, 48, 110, 10, 26, 30,4, 8, 22, 20, 18, 14]
```

```
Profits = [25, 18, 27, 20, 23, 17, 21, 17, 15, 13, 9, 14, 11, 10, 12, 8, 7, 5, 3, 1]
```

Il faut ensuite itérer les opérations suivantes jusqu'au dernier élément :

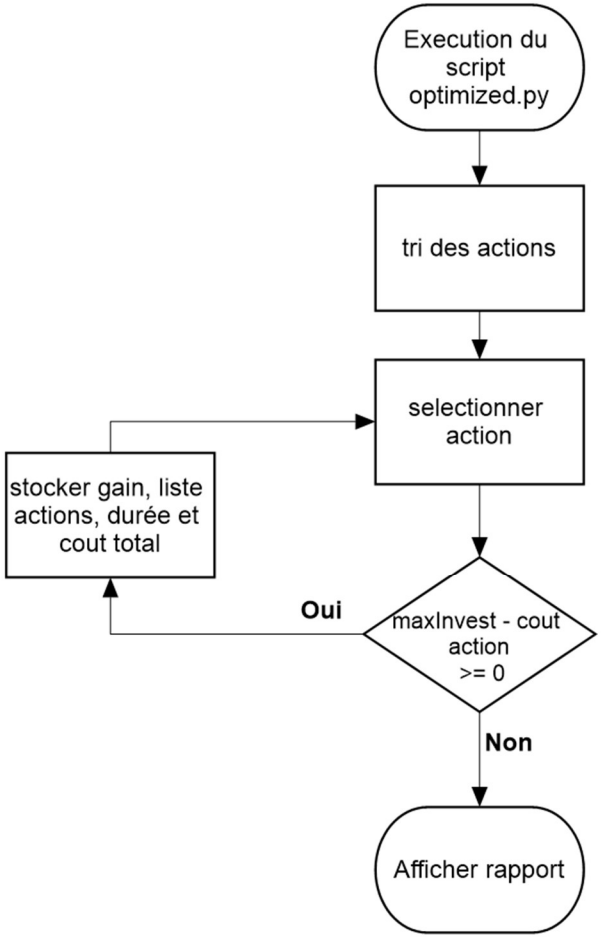
1) Additionner un à un le coût des actions en suivant l'ordre des éléments de la liste COSTS trié sans jamais dépasser 500€
2) Additionner le gain (gain = coût x (profit / 100)) associé à l'action sélectionnée

Le calcul de complexité temporelle est linéaire donc O(n), « n » étant le nombre d'actions qu'il est possible d'acheter. Dans l'hypothèse où la solution optimale serait l'achat de toutes les actions, l'itération se produirait donc autant de fois qu'il y a d'article. Dans cette hypothèse, la complexité temporelle serait donc égale à O(20). La complexité spatiale étant quant à elle « n » (l'espace mémoire utilisé sera toujours lié au nombre d'action dans le dataset et est donc linéaire).

Pseudo-code :

```
Algorithme glouton

Variable

	actionsSorted : LISTE CONTENANT LES OBJETS TRIES (OBJET = NOM, COUT, PROFIT ET RATIO)

	actionsNames <- ‘’ : VARIABLE VIDE QUI CONTIENDRA LA LISTE DES ACTIONS A ACHETER

	finalProfit <- 0 : VARIABLE VIDE QUI CONTIENDRA LE GAIN TOTAL

	maxInvest <- 500 : VARIABLE CONTENANT LE COUT MAXIMUM A NE PAS DEPASSER

Début

	POUR i <- 0 ; actionsSorted; i = i + 1

		SI maxInvest - cout(actionsSorted[i]) >= 0 :

			maxInvest = maxInvest – cout(actionSorted[i])

			finalProfit = cout(actionSorted[i]) x profit(actionSorted[i] / 100)

			actionsNames = actionsNames + nom(actionSorted[i])

		Fin SI

	Fin POUR

Fin
```

Organigramme :



Depuis dossier *controllers*: `python.exe .\optimized.py ..\datas\dataset.csv`

`actionsSorted.sort(reverse=True)`

boucle `for action in actionsSorted:`

```
finalProfit += cost * profit / 100
actionsNames += action.name + ", "
str(datetime.datetime.now() - start
MAX_INVEST - maxInvest
```

`if maxInvest - cost >= 0:`

`optimizedReport(getMaxProfit(MAX_INVEST, actionsObjList ))`

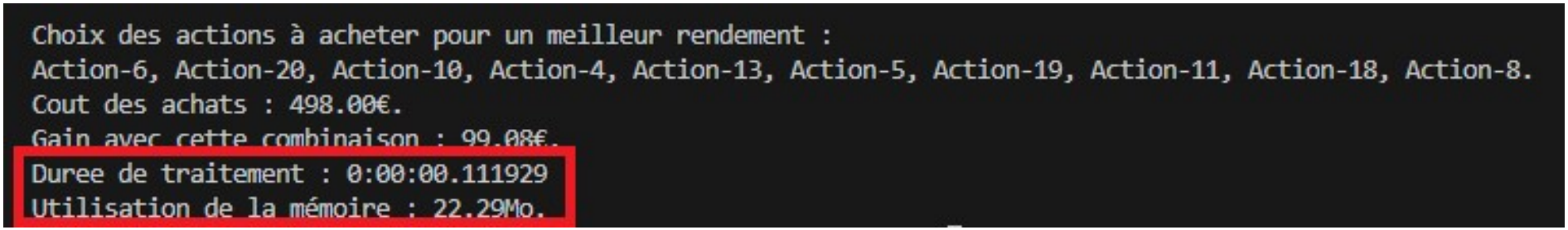
Conclusion :

Avantage : temps de traitement.

Inconvénient : toutes les combinaisons ne sont pas testées. Cet algorithme additionne en priorité les actions ayant le meilleur rendement.

Cette méthode heuristique tombe sur le même résultat que la méthode brute force. Malgré tout, elle risque de ne pas être exacte dans d'autres situations, mais pourrait donc servir pour des demandes déterminant une tendance par exemple. Dans ce cas précis, le gros intérêt serait donc la rapidité d'obtention d'une réponse.

Le temps d'exécution est d'environ un dixième de seconde et l'espace mémoire occupé est petit aussi vu le faible nombre de variables utilisées. Ces valeurs s'expliquent par les faibles valeurs de la complexité temporelle et spatiale.



Algorithme dynamique

Cette méthode utilise la récursivité combinée à la comparaison d'anciennes valeurs de gain pour tester l'ensemble des combinaisons possibles.

Tout d'abord créer un tableau de dimension [INVESTISSEMENT_MAX] X [NOMBRE_ACTIONS] soit un tableau de 500 X 20.

L'itération consiste à tester les gains pour chacune des actions, comparés aux gains des précédentes pour déterminer la combinaison qui rapporte le plus :

- Si la combinaison gain + « gain de la valeur restante précédente » est supérieure au gain précédent pour le même montant, on conserve ce résultat
- Si la combinaison gain + « gain de la valeur restante précédente » est inférieur au gain précédent pour le même montant rapporte moins, on conserve la valeur du gain précédent

Exemple : admettons que nous procédons à l'achat d'actions pour un montant de **345€**. Considérons l'achat d'une action de **225€** et qui rapporte 20€ et la valeur du gain du montant restant 120€ (**345€** – **225€**) qui vaut 7€. Si l'addition de ces deux montants vaut plus que le gain associé à l'achat de l'action précédente pour le même montant (**345€**), on conserve la combinaison qui donne le gain le plus élevé, sinon on conserve le gain lié à l'achat de l'action précédente pour le même montant (**345€**).

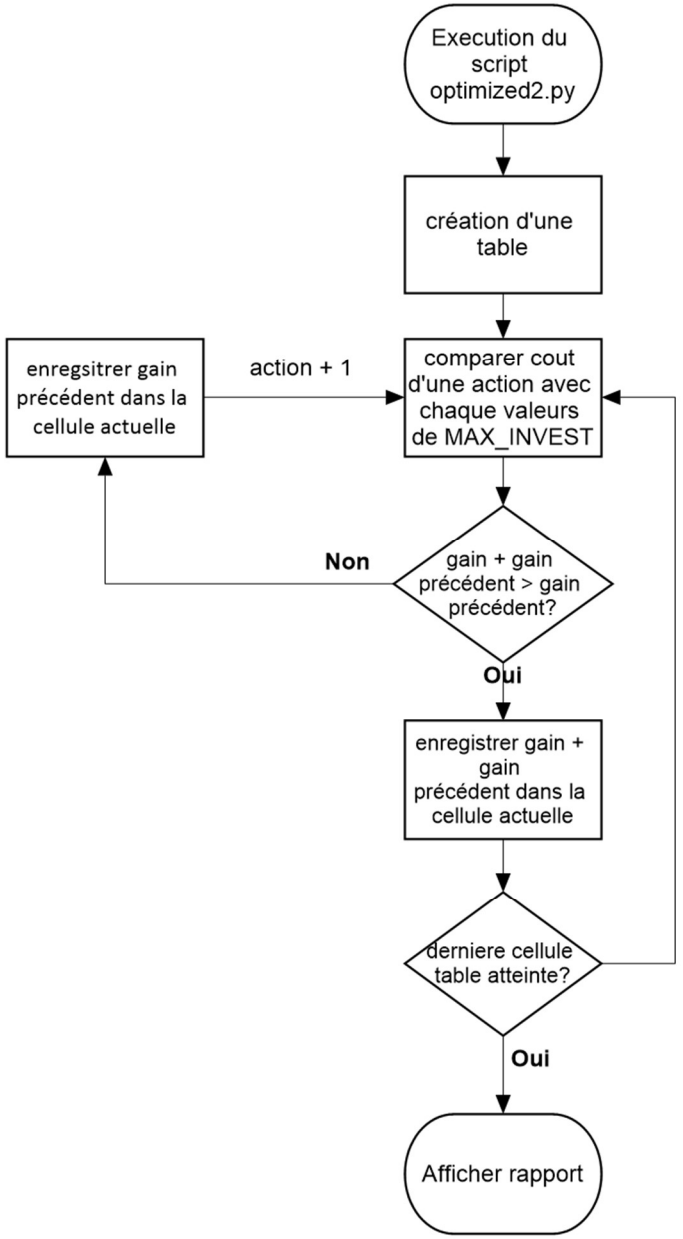
Le calcul de complexité temporelle est 0(n*m), « n » étant égal au nombre d'actions qu'il est possible d'acheter et « m » le montant maximum des achats qu'il est possible d'effectuer. Dans notre contexte, ce sera donc O(20*500) c'est-à-dire O(10000). Etant donné que pour déduire la meilleure combinaison nous n'avons pas d'autre choix que de toutes les tester, cela rend constant le temps de traitement donc dans notre contexte, nous pouvons en déduire que le calcul de complexité temporel sera égal à O(10 000). A noter également que l'algorithme dynamique ne peut s'utiliser avec des valeurs négatives, car ces mêmes valeurs servent à « adresser la matrice », c'est-à-dire sélectionner une cellule par ses coordonnées de colonne et cellule. Dans un tel cas, il est donc nécessaire de multiplier les valeurs par un coefficient équivalent au nombre de décimale du chiffre en ayant le plus. Dans notre contexte, la majeure partie des actions ont des coûts possédant deux décimales. Il conviendra donc de multiplier « n » et « m » par 100 sans oublier de diviser les résultats de gain et profit par le même coefficient.

Le calcul de complexité spatiale est donc n*m également (« n*m » étant le nombre de cases à remplir dans le tableau et donc au nombre de variable de fois où les variables utilisées seront initiées). Dans notre contexte le calcul donnera donc 10 000 car l'itération se produira forcément 10 000 fois avant de produire un résultat (le gain de la meilleure combinaison se trouvera dans la toute dernière cellule du tableau, qui elle-même ne peut être calculée que grâce à toutes les autres).

Pseudo-code :

```
Algorithme dynamique
Variable
	actions <- OBJETS CONTENANT LES ACTIONS (CONTIENT NOM, COUT et PROFIT)
	MAX_INVEST <- 500 (INVESTISSEMENT MAXIMUM)
	table : TABLEAU DE DIMENSION « LEN(NOMBRE D'ACTIONS) * LEN(MAX_INVEST) »
	actionsToBuy <- [] : LISTE CONTENANT LA MEILLEURE COMBINAISON D'ACTION A ACHETER
Début
	Remplir les premières lignes et colonnes de la table avec des « 0 »
	POUR i <- 1 ; actions ; i = i + 1
		POUR j <- 1 ; MAX_INVEST ; j = j + 1
			SI cout action[i] <= j :
				SI gain action[i] + gain montant restant à investir > gain de l'action précédente pour même montant
					Enregistrer le gain dans la cellule en cours
				SINON
					Enregistrer le gain de l'action précédente dans la cellule en cours
				Fin SI
			SINON
				Enregistrer le gain de l'action précédente dans la cellule en cours
			Fin SI
		Fin POUR
	Fin POUR
```

Organigramme :



Depuis dossier *controllers*: `python.exe .\optimized2.py ..\datas\dataset.csv`

```
table = [[0 for x in range(maxInvest + 1)] for x in range(n + 1)]
```

boucle `if cost <= j:`

```
table[i][j] = table[i - 1][j]
```

```
if gain + table[i - 1][j - int(cost)] > table[i - 1][j]:
```

```
table[i][j] = gain + table[i - 1][j - int(cost)]
```

```
optimizedReport(getMaxProfit(maxInvest, actionsObjList, actionsContainFloat))
```

Conclusion :

Avantage : précision du résultat, car beaucoup de combinaisons sont testées.

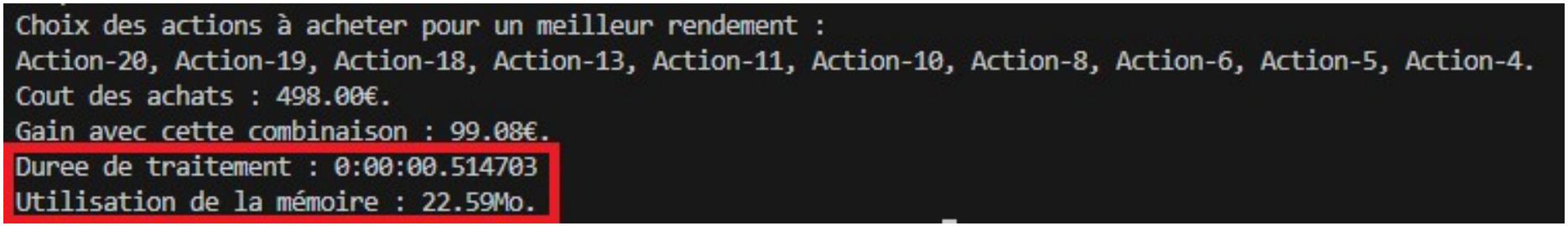
Inconvénient : la ressource processeur est plus élevé, car beaucoup plus de traitement que la « version glouton » : O(10 000) > O(20)

De plus, le traitement est différent et propose une autre combinaison (malgré un résultat cout et gain identique, la combinaison d'achat des actions est différente). Cette solution permet, d'après moi, d'avoir un résultat plus précis (et non plus une tendance) car elle compare beaucoup de combinaisons. Mais elle aura tendance à prendre de plus en plus de temps en fonction du nombre d'éléments à tester, ce qui en est donc ses limites.

Etant donné que la complexité temporelle de ce script est linéaire (O(n)), plus le fichier en entrée est gros, plus le temps de traitement est long ce qui en est donc ses limites. Cela explique la durée d'environ 30 secondes. De plus, ce script utilise une matrice. Les matrices prennent beaucoup de place en mémoire ce qui explique la quantité de mémoire plus élevé qu'avec les autres scripts. Dans notre cas, 1000 actions sont traitées.

Remarque :

Ci-dessous, comparativement, avec le fichier contenant 20 actions, on obtient un temps de traitement beaucoup plus court et un espace mémoire occupé beaucoup plus faible. Ce qui met en évidence la linéarité de la complexité temporelle et spatiale de ce script car moins d'actions = moins de traitement et moins de besoin en espace mémoire car moins de variables à initier.



Partie 3 : backtesting et optimisation

Sienna de la société *AlgoInvest&Trade* demande à ce que les scripts soient exécutés sur d'anciennes décisions d'achats d'actions afin de déterminer la fiabilité des résultats des scripts et de ses décisions.

Tout d'abord et suite à l'étude des dataset fournis par *Sienna*, il s'avère que certaines données sont incorrectes : certaines actions ont un coût négatif. A contrario d'un profit négatif (significatif d'un mauvais rendement), un coût négatif n'est, à mon sens, pas possible. J'ai donc décidé d'intégrer dans mes scripts une vérification de ces actions incorrectes et de les mettre à zéro au niveau coût et profit afin qu'ils n'entrent pas en compte dans le calcul.

Cela se passe au niveau du constructeur, dans la *Class porteFolio* du module *models\actions.py*:

```
class porteFolio:
    def __init__(self, name, cost, profit, actionsContainFloat):
        self.name = name
        self.cost = float(cost)
        # if cost <= 0 (so I guess it's a mistake), I modify cost and gain by 0 to bypass this action
        if self.cost <= 0:
            self.cost = 0
            self.profit = 0
        else:
            self.profit = float(profit)
        if actionsContainFloat:
            self.gain = self.cost * (self.profit / 10000)
        else:
            self.gain = self.cost * (self.profit / 100 )
        self.ratio = (self.cost * self.profit / 100) + self.profit
```

Ci-dessous le tableau comparatif des algorithmes ainsi que la comparaison des propositions de Sienna :

				Glouton *optimized.py*	***Algo dynamique*** *optimized2.py*	***Bruteforce*** *bruteforce.py*
Datasets	dataset.csv 20 actions		***Temps de traitement***	0.11 sec	0.51 sec	8.62 sec
			Mémoire utilisée	22.29Mo	22.59Mo	22.18Mo
			Coût	498.00	498.00 €	498.00
			Profit	99.08	99.08 €	99.08
			Ratio	19.90%	19.90%	19.90%
	dataset1_Python+P7.csv 1000 actions		***Tps traitement***	0.11 sec	29.25 sec	trop long
			Mémoire	22.87Mo	55.93Mo	-
		mon script	***Coût***	499.99 €	499.96 €	-
			Profit	197.09 €	198.55 €	-
			Ratio	39.42%	39.71%	-
		Sienna	***Coût***	498.76 €		-
			Profit	196.61 €		-
			Ratio	39.42%		-
	dataset2_Python+P7.csv 1000 actions		***Tps traitement***	0.11 sec	16.15 sec	trop long
			Mémoire	22.64Mo	40.21Mo	-
		mon script	***Coût***	499.77 €	499.92 €	-
			Profit	192.75 €	197.96 €	-
			Ratio	38.57%	39.60%	-
		Sienna	***Coût***	489.24 €		-
			Profit	193.78 €		-
			Ratio	39.61%		-

Cela met clairement en évidence deux choses :

- En ne prenant pas en compte les actions avec coûts négatifs, on retombe sur les mêmes valeurs que Sienna (valeurs en bleu et violets). Ce qui suppose que les actions avec coûts négatifs ont été intégrées au fichier dataset par erreur puisque Sienna elle-même ne les avait pas intégrés lors de ses calculs.
- Le meilleur résultat semble être fourni par le script à base d'algorithme dynamique. Avec ce script, soit on obtient le même résultat que Sienna, soit un meilleur résultat, c'est-à-dire un ratio profit/achat supérieur.

Conclusion finale :

Il est important de se demander ce que souhaite le client, si l'exactitude des résultats est exigée ou bien si une un tendance (précise malgré tout, bien sur) suffit. A la lecture des échanges entre le client et la société, il en ressort deux éléments :

- Le client est pressé

> Il semble que le programme que vous avez créé prenne beaucoup de temps et de ressources pour choisir la meilleure opportunité d'investissement. Nos clients sont des personnes très occupées, et préfèrent une production beaucoup plus rapide. Le programme doit fournir une réponse en moins d'une seconde.

- Il n'est pas utile de tester toutes les combinaisons, on peut donc en déduire que si on ne teste pas toutes les combinaisons, il se peut que le résultat ne soit donc pas optimum car cela ne fera donc pas ressortir à coup sur la meilleure combinaison parmi toutes les possibilités.

> Nous avons discuté avec nos clients et ils font confiance à AlgoInvest&Trade pour prendre les bonnes décisions. Nous n'avons plus besoin d'explorer toutes les combinaisons possibles, ce qui, à mon avis, devrait accélérer l'algorithme.

Ces éléments de réponses ont tendance à s'orienter vers le choix d'un script rapide et avec une précision de résultat s'approchant au maximum du meilleur. J'en conclus donc que le script « glouton » serait, dans ce contexte, le plus approprié. Les résultats de ce script étant soit identique, soit sensiblement inférieur aux résultats de Sienna lors du *backtesting*.