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 une à une l'ensemble des combinaisons d'actions pour en déterminer celle qui a le meilleur profit.

<u>Calcul du nombre de possibilités</u> :

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

<u>Proposition d'algorithme</u> :

A partir d'une fonction calculant toutes les combinaisons possibles, sélectionner celles dont le coût ne dépasse pas 500€.

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

Si supérieur : stockage de la combinaison d'action dans une liste et des valeurs de coût et rendement pour la génération d'un rapport.

Pour faire le choix des actions à acheter, j'utilise un « compteur binaire » sur n digits (« n » étant égal au nombre d'actions) :

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.

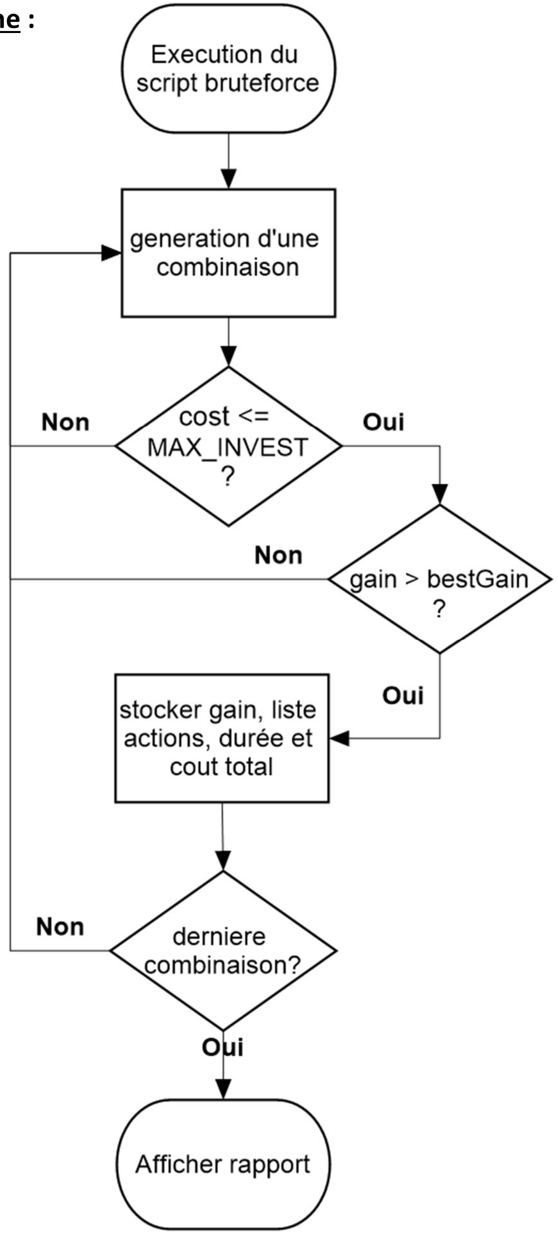
Le calcul de complexité temporelle est linéaire donc $O(2^n)$. Etant donné que pour déduire la meilleure combinaison nous n'avons pas d'autres choix que de toutes les tester, cela rend constant le temps de traitement donc dans notre contexte, nous pouvons en déduire que le calcul de complexité temporelle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

La complexité spatiale est 2^n également car comme dit précédemment et dans notre contexte, pas d'autres choix que de tester toutes les combinaisons donc les variables seront initiées 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. Le résultat final est donc d’une précise très fine.

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 (moins de 10 secondes) car très peu d’actions à tester. Idem pour l’espace mémoire utilisé, car la boucle ne se répète pas beaucoup de fois « du point de vue d’un microprocesseur ».

```
Choix des actions à acheter pour un meilleur rendement :
Action-4, Action-5, Action-6, Action-8, Action-10, Action-11, Action-13, Action-18, Action-19, Action-20.
Cout des achats : 498.0€.
Gain avec cette combinaison : 99.08€.
Duree de traitement : 0:00:08.62
Utilisation de la mémoire : 22.18Mo.
```

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 :

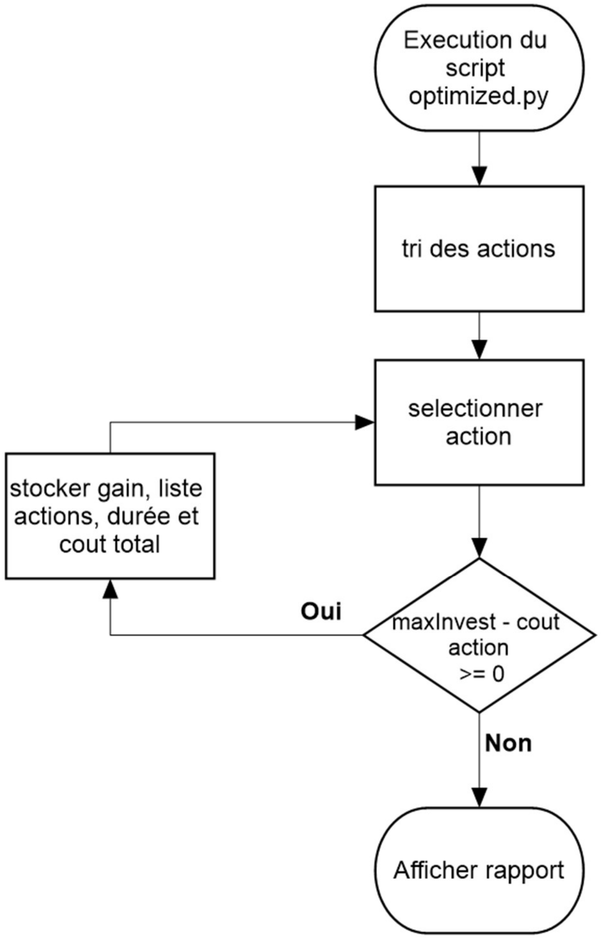
1) Additionner un à un le coût des actions en suivant l'ordre des éléments de la liste COSTS trié sans jamais dépasser 500€
2) Additionner le gain (gain = coût x (profit / 100)) associé à l'action sélectionnée

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 est également linéaire donc « n » car l'itération initiant les variables se produira n fois.

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
```

<u>Organigramme</u> :



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. Une fois atteint ou dépassé, le script s'arrête et n'essaye pas d'autres combinaisons.

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 notamment quand il y aura plus de choix et donc plus de combinaisons d'actions à acheter. Malgré tout cette méthode pourrait servir pour des demandes déterminant une tendance par exemple. Dans ce cas précis, le gros intérêt serait donc la rapidité d'obtention d'un résultat.

Le temps d'exécution est d'environ trois centième 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.

```
Choix des actions à acheter pour un meilleur rendement :
 Action-6, Action-20, Action-10, Action-4, Action-13, Action-5, Action-19, Action-11, Action-18, Action-8.
Cout des achats : 498.00€.
Gain avec cette combinaison : 99.08€.
Duree de traitement : 0:00:00.036308
Utilisation de la mémoire : 22.34Mo
```

Algorithme dynamique

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

Tout d'abord créer un tableau de dimension [investissement maximum] X [nombre d'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 du montant restant » est supérieure au gain précédent pour le même montant, on conserve ce résultat
- Si la combinaison gain + « gain du montant restant » est inférieur au gain précédent pour le même montant, 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€ (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. 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. Dans notre contexte donc, nous pouvons en déduire que le calcul de complexité temporel sera égal à O(20*500) soit : O(10 000).

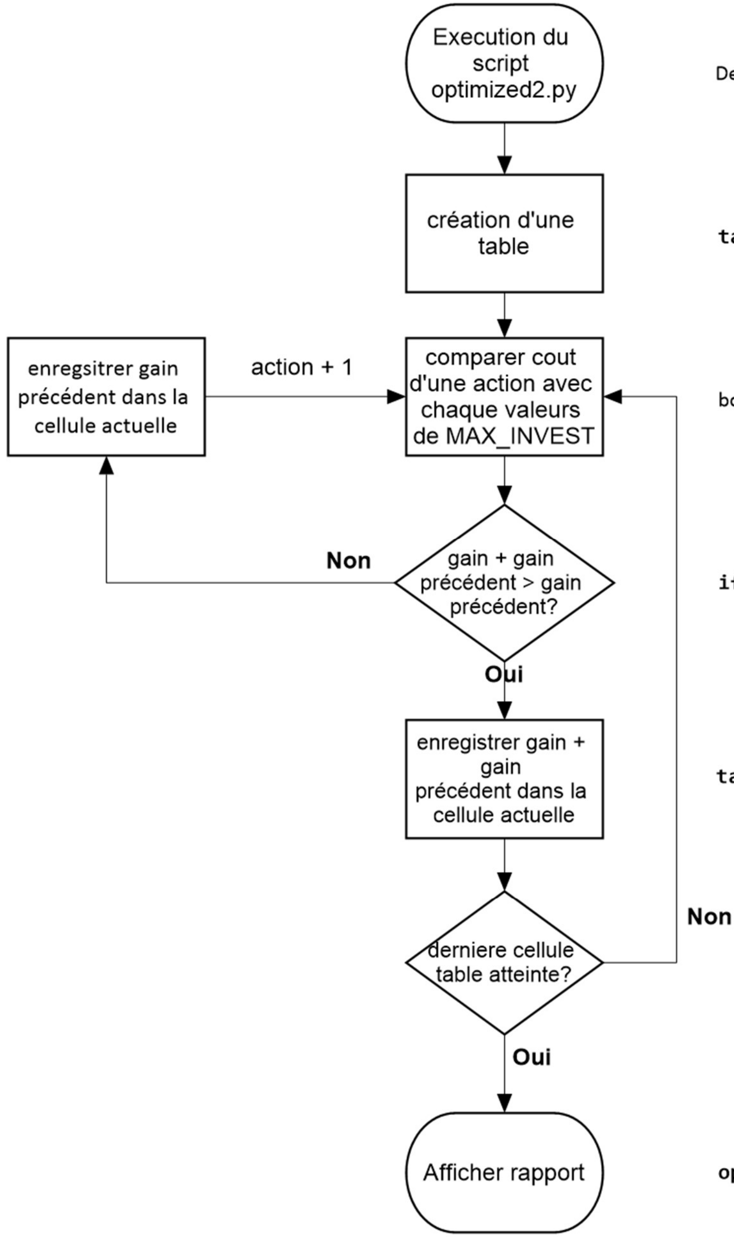
A noter également que l'algorithme dynamique ne peut s'utiliser avec des valeurs négatives, car ces mêmes valeurs servent à « adresser la matrice », c'est-à-dire sélectionner une cellule avec ses coordonnées de colonne et cellule. Dans un tel cas, il est donc nécessaire de multiplier les valeurs par un coefficient égal au nombre de décimale du nombre en ayant le plus et l'appliquer à l'ensemble des valeurs du fichier. Dans notre contexte, la majeure partie des actions ont des coûts possédant deux décimales. Il conviendra donc de multiplier les coûts et profits par 100 sans oublier de diviser ces mêmes valeurs par le même coefficient en fin de calcul.

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 fois où les variables utilisées seront initiées). Dans notre contexte le calcul de complexité temporel donnera donc 10 000 (20*500) car les variables seront initiées autant de fois qu'il y aura d'itération avant de produire un résultat final (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)

    maxInvest <- 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 ; maxInvest; 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
```

<u>Organigramme</u> :



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

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

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

boucle `if cost <= 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 : les ressources processeur et mémoire nécessaires au calcul sont forcément plus élevées que celle de l'algorithme glouton car l'élément supplémentaire « m » entre compte dans le calcul de complexité temporel et spatial.

Cette solution permet, d'après moi, d'avoir un résultat plus précis 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.

Cela explique également la durée de traitement supérieur à celle de l'algorithme glouton (une demie seconde contre trois centièmes).

```
Choix des actions à acheter pour un meilleur rendement :
 Action-20, Action-19, Action-18, Action-13, Action-11, Action-10, Action-8, Action-6, Action-5, Action-4.
Cout des achats : 498.00€.
Gain avec cette combinaison : 99.08€.
Duree de traitement : 0:00:00.566673
Utilisation de la mémoire : 22.74Mo
```

Remarque :

Dans la suite du projet, Sienna fournit des fichiers contenant beaucoup d'actions. C'est l'occasion de mettre en évidence l'augmentation du calcul de complexité temporel et spatial. Ce qui s'explique logiquement car O(1000*500) > O(20*500) et le nombre de variables initiées s'en trouve également lourdement augmenté. Le temps de traitement et l'espace mémoire augmentent donc considérablement.

```
Choix des actions à acheter pour un meilleur rendement :
 Share-KMTG, Share-GHIZ, Share-NHWA, Share-UEZB, Share-LPDM, Share-MTLR, Share-USSR, Share-GTQK, Share-FKJW,
Share-GIAJ, Share-XJMO, Share-LRBZ, Share-KZBL, Share-EMOV, Share-IFCP.
Cout des achats : 499.96€.
Gain avec cette combinaison : 198.55€.
Duree de traitement : 0:00:27.780525
Utilisation de la mémoire : 52.01Mo
```

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 d'en déterminer la fiabilité.

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 inférieur ou égal à 0. A contrario d'un profit négatif (significatif d'un mauvais rendement), un coût nul ou 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 exclure du calcul. Pour les identifier, je configure le coût à 0 pour qu'elles soient identifiables dans la suite du script.

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 by 0 to bypass this action in the script
        if self.cost <= 0:
            self.cost = 0
        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 :

<table>
<tr><th colspan="4"></th><th>Glouton
optimized.py</th><th>Algo dynamique
optimized2.py</th><th>Bruteforce
bruteforce.py</th></tr>
<tr><td rowspan="28">Datasets</td><td rowspan="5">dataset.csv
20 actions</td><td colspan="2">Temps de traitement</td><td>0.036 sec</td><td>0.51 sec</td><td>8.62 sec</td></tr>
<tr><td colspan="2">Mémoire utilisée</td><td>22.34Mo</td><td>22.59Mo</td><td>22.18Mo</td></tr>
<tr><td colspan="2">Coût</td><td>498.00 €</td><td>498.00 €</td><td>498.00 €</td></tr>
<tr><td colspan="2">Profit</td><td>99.08 €</td><td>99.08 €</td><td>99.08 €</td></tr>
<tr><td colspan="2">Ratio</td><td>19.90%</td><td>19.90%</td><td>19.90%</td></tr>
<tr><td rowspan="8">dataset1_Python+P7.csv
1000 actions</td><td colspan="2">Temps de traitement</td><td>0.04 sec</td><td>29.25 sec</td><td>trop long</td></tr>
<tr><td colspan="2">Mémoire</td><td>22.87Mo</td><td>55.93Mo</td><td>-</td></tr>
<tr><td rowspan="3">mon script</td><td>Coût</td><td>499.99 €</td><td>499.96 €</td><td>-</td></tr>
<tr><td>Profit</td><td>197.09 €</td><td>198.55 €</td><td>-</td></tr>
<tr><td>Ratio</td><td>39.42%</td><td>39.71%</td><td>-</td></tr>
<tr><td rowspan="3">Sienna</td><td>Coût</td><td colspan="2">498.76 €</td><td>-</td></tr>
<tr><td>Profit</td><td colspan="2">196.61 €</td><td>-</td></tr>
<tr><td>Ratio</td><td colspan="2">39.42%</td><td>-</td></tr>
<tr><td rowspan="8">dataset2_Python+P7.csv
1000 actions</td><td colspan="2">Temps de traitement</td><td>0.037 sec</td><td>16.15 sec</td><td>trop long</td></tr>
<tr><td colspan="2">Mémoire</td><td>22.64Mo</td><td>40.21Mo</td><td>-</td></tr>
<tr><td rowspan="3">mon script</td><td>Coût</td><td>499.77 €</td><td>499.92 €</td><td>-</td></tr>
<tr><td>Profit</td><td>192.75 €</td><td>197.96 €</td><td>-</td></tr>
<tr><td>Ratio</td><td>38.57%</td><td>39.61%</td><td>-</td></tr>
<tr><td rowspan="3">Sienna</td><td>Coût</td><td colspan="2">489.24 €</td><td>-</td></tr>
<tr><td>Profit</td><td colspan="2">193.78 €</td><td>-</td></tr>
<tr><td>Ratio</td><td colspan="2">39.61%</td><td>-</td></tr>
</table>

Rouge = valeurs trop grandes / bleu et violet valeurs identiques à Sienna

On constate qu'en ne prenant pas en compte les actions avec coûts négatifs ou égal à zéro, 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.

Conclusion finale :

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

- 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.

- Toutes les combinaisons n'ont pas besoin d'être testées :

> 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 d'informations ont tendance à s'orienter vers le choix d'un script rapide et avec une précision de résultat s'approchant au maximum du meilleur sans pour autant l'atteindre absolument. 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* mais aussi obtenu dans un temps record.