AlgoInvest&Trade

La société *AlgoInvest&Trade* propose à ses clients des choix d’actions à acheter pour en tirer le meilleur profit.

Plusieurs contraintes sont appliquées pour faire ces choix :

* Une action peut être achetée qu’une seule fois
* Une action ne peut se vendre fractionnée
* Total d’achat maximum 500€

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

Bruteforce

**Calcul du nombre de possibilités :**

* 20 actions comprenant 2 possibilités :
  + Acheter l’action
  + Ne pas acheter l’action

Donc 220 = 1 048 576 possibilités

**Proposition d’algorithme** :

A partir de la liste *(listActions[220])* de toutes les combinaisons possible, sélectionner celles dont le cout ne dépasse pas 500€.

Pour chacune de ces lignes, calculer le rendement et vérifier si supérieur à la valeur précédente.

Si supérieur : stockage de la combinaison d’action dans une liste (*meilleurCombinaisonActions[]*) et stocker 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 | 0 | 0 |
| **N° action** | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

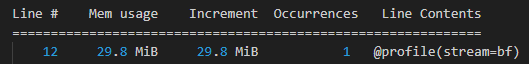
Cette combinaison représente donc l’achat des actions 19, 18, 14, 13, 9, 7, 6 et 5.

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

Le calcul de complexité temporelle est 0(2n). 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 temporelle sera égale en permanence à O(220) soit 1 048 576 combinaisons.

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

La place mémoire qui en résulte est de 29.8 Mo :



**Pseudo-code :**

Algorithme meilleurRendement

Variable

listActions[220] : LISTE CONTENANT TOUTES LES COMBINAISONS DE SELECTION

meilleurCombinaisonActions[] : LISTE VIDE POUR STOCKER LA COMBINAISON D’ACTION FINALE A ACHETER

meilleurRendement <- 0 : RENDEMENT ASSOCIE A LA MEILLEURE COMBINAISON D’ACTION

Début

Pour i <- 1 ; taille listCombinaisonActions[220] ; i = i + 1

SI cout(listCombinaisonActions[i]) =< 500 :

SI rendement(listCombinaisonActions[i]) > meilleurRendement :

meilleurCombinaisonActions[] <- listCombinaisonActions[i]

meilleurRendement <- rendement(listCombinaisonActions[i])

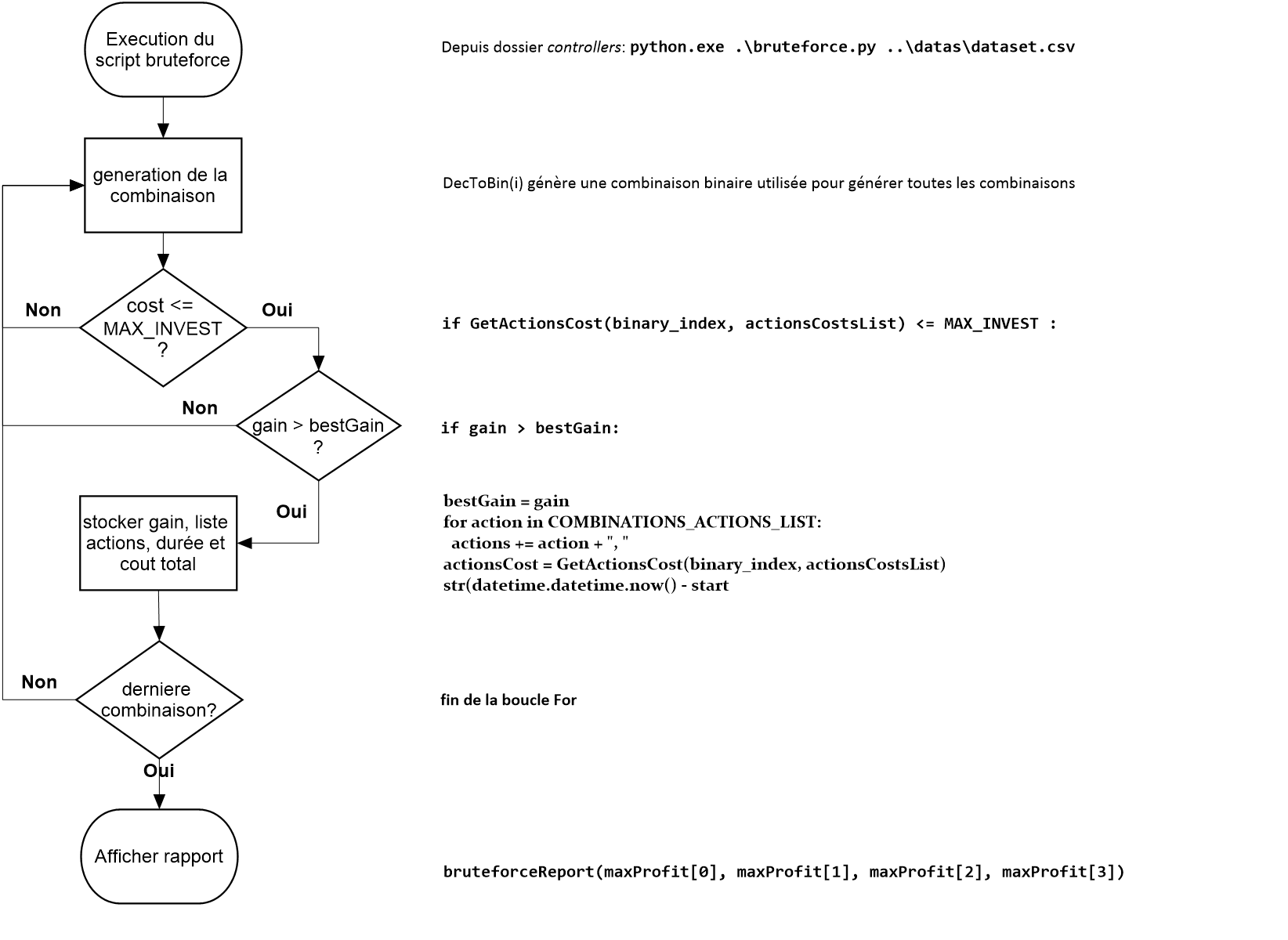
Fin SI

Fin SI

Fin POUR

Fin

**Organigramme :**

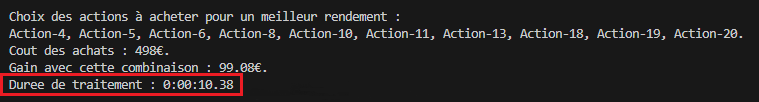
****

**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 impacté. Si nous ajoutons un choix supplémentaire de 5 actions à acheter, la complexité temporelle passera de O(220) à O(225), le temps de traitement en sera donc lourdement allongé. 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.



Optimisation

Methode 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 fournis 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 donc itérer les opérations suivantes jusqu’au dernier élément :

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

Le calcul de complexité temporelle est 0(n), « n » étant le nombre d’actions qu’il est possible d’acheter, contenu dans le dataset. 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 constante et donc égale à 0(20).

La complexité spatiale étant quant à elle « n » (l’espace mémoire maximal utilisé sera au pire le nombre d’action disponible dans le dataset)

La place mémoire occupée est de 29.8Mo dans notre contexte est :



**Pseudo-code :**

Algorithme glouton

Variable

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

actionsNames <- ‘’ : VARIABLE CONTENANT LISTE DES ACTIONS A ACHETER

finalProfit <- 0 : VARIABLE CONTENANT LE GAIN TOTAL

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

Début

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

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

maxInvest = maxInvest – cout(actionSorted[i])

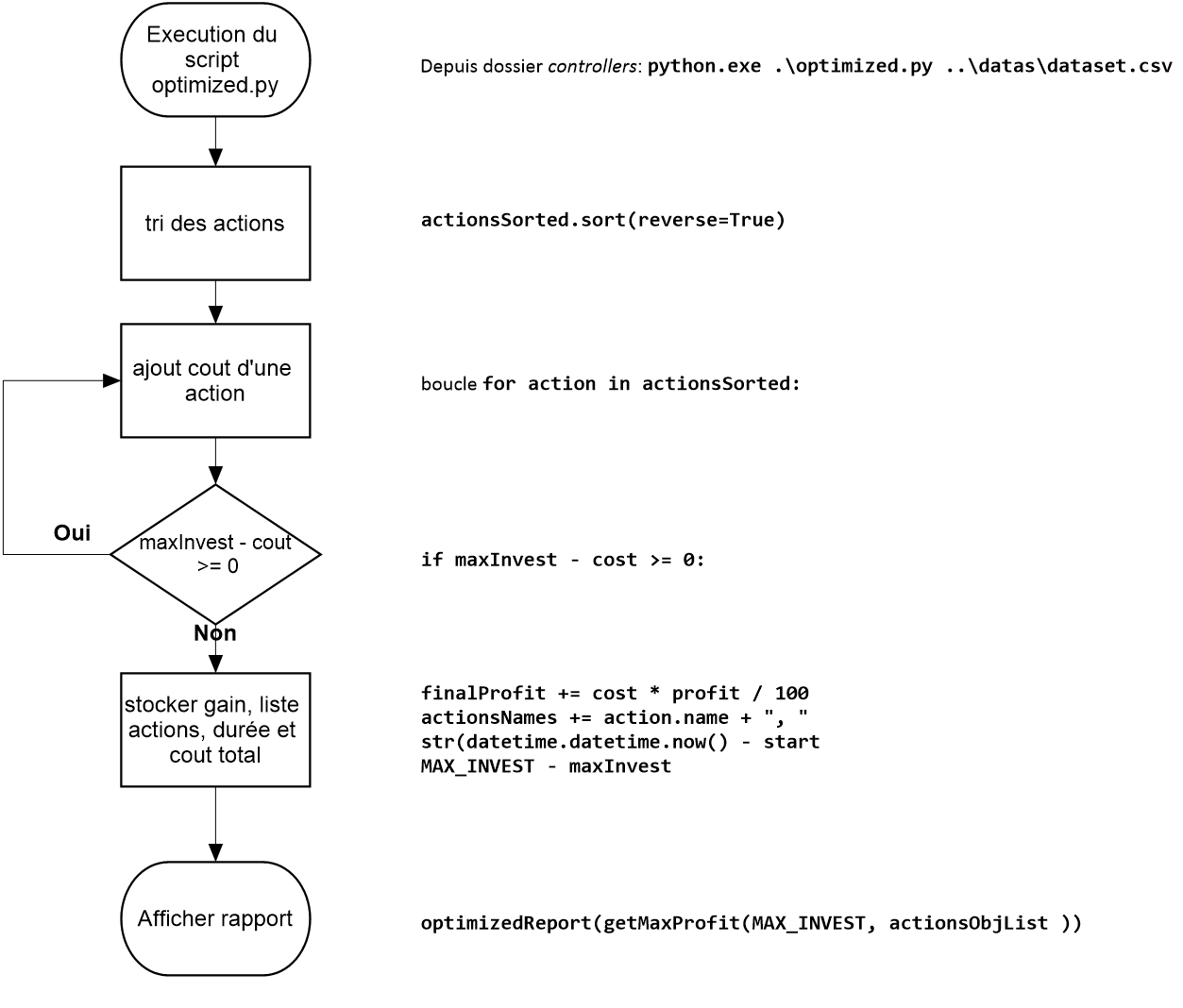
finalProfit = cout(actionSorted[i]) x profit(actionSorted[i])

actionsNames = actionsNames + nom(actionSorted[i])

Fin SI

Fin POUR

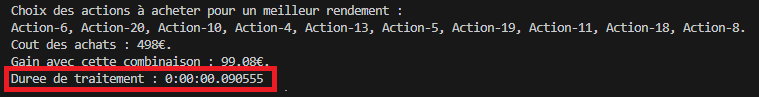
Fin

**Organigramme** :

**Conclusion** :

Avantage : temps de traitement.

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



Optimisation

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][NOMBRE\_ACTIONS] soit un de 500 (colonnes) X 20 (lignes)

L’itération consiste à tester les gains pour chacune des actions comparé aux gains des précédentes pour determiner la combinaison qui rapporte le plus :

* Si la combinaison gain + gain de la valeur restante précédente est supérieur 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 de la valeur restante 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 précédent 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), « n » étant égal au nombre de combinaison possible c’est-à-dire la taille du tableau créé au départ de l’algorithme : 20 x 500 soit 10 000. 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 égale en permanence à O(10 000).

Le calcul de complexité spatiale est donc n, (« n » étant le nombre de case à remplir dans le tableau). Dans notre contexte le calcul donnera donc 10 000 car l’itération se produira forcement 10 000 fois avant de produire une conclusion.

**Pseudo-code** :

Algorithme dynamique

Variable

actions <- OBJETS CONTENANT ACTIONS(CONTIENT NOM, COUT et PROFIT)

MAX\_INVEST <- 500 (INVESTISSEMENT MAXIMUM)

table : TABLEAU DE DIMENSION « LEN(actionsObjList) \* LEN(MAX\_INVEST) »

Début

Remplir premières lignes et colonnes de table avec des « 0 »

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

POUR j <- 0 ; MAX\_INVEST ; j = j + 1

SI cout action[i] <= j :

SI gain action[i] + gain montant restant à investir (table[i-1][j-cout action[j])

Enregistrer dans cellule en cours de table

SINON

Enregistrer dans cellule gain de l’action précédent

Fin SI

SINON

Enregistrer dans cellule gain de l’action précédent

Fin SI

Fin POUR

Fin POUR

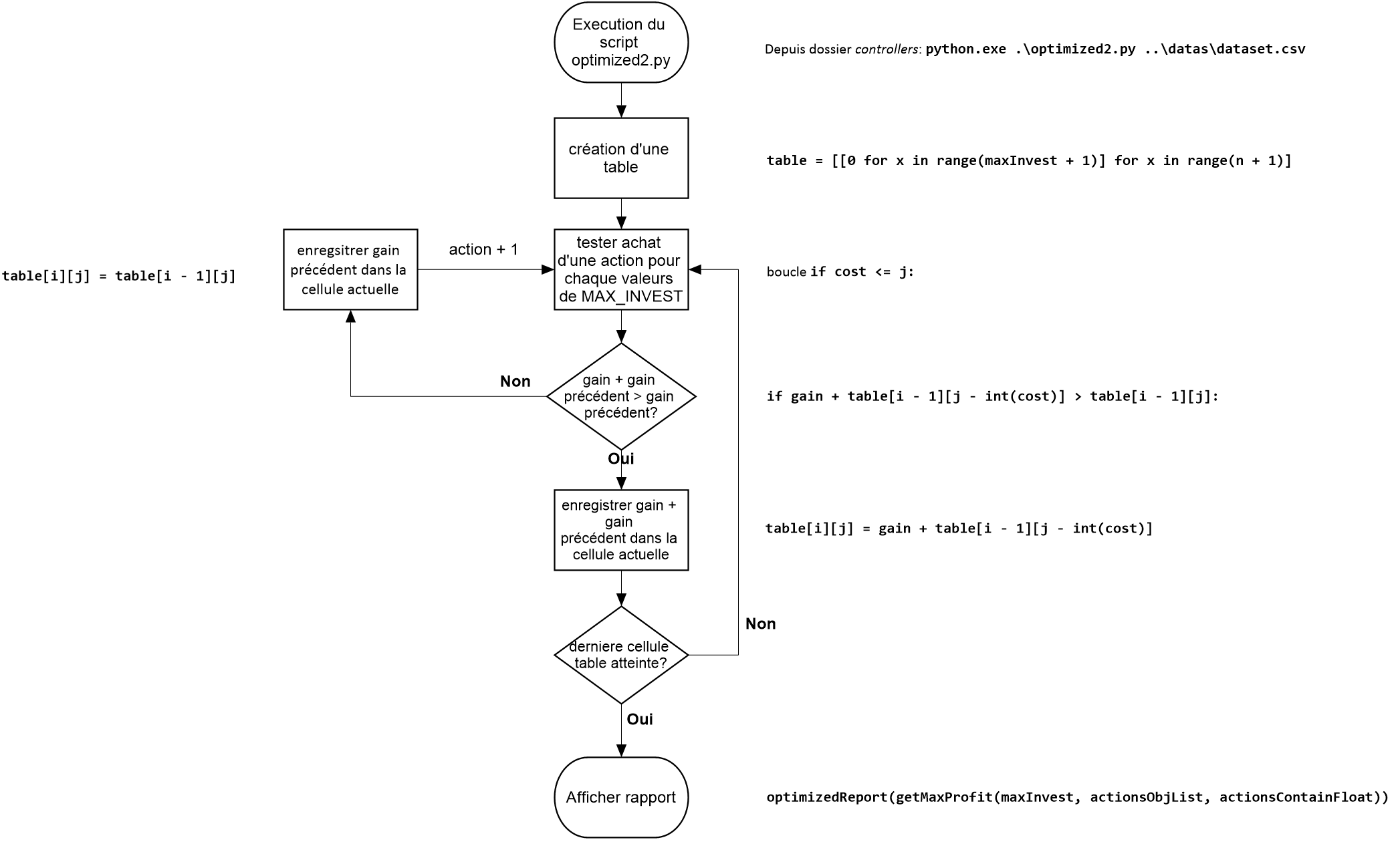
Afficher dernière cellule (table[len(actions)][MAX\_INVEST])

Constituer listing des actions à acheter et stocker dans liste actionsToBuy

Calculer total cout

Afficher rapport

Fin

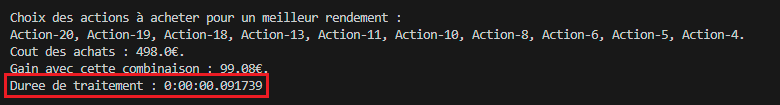
**Organigramme :**

**Conclusion** :

Avantage : temps de traitement, quasi idem que la « version glouton ».

Inconvénient : ressource processeur plus élevé car beaucoup plus de traitement que la « version glouton » : 0(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).



La place mémoire occupée est de 30Mo :



backtesting et

optimisation