

# ALGO INVEST

## SOMMAIRE

- ALGORITHME DE FORCE BRUTE
- L'ALGORITHME GROUTON (SOLUTION OPTIMISÉE)
- L'ANALYSE DES PERFORMANCES DES ALGORITHMES
- COMPARAISON ENTRE LES RÉSULTATS DE L'ALGORITHME ET LES CHOIX DE SIENNA
- RAPPORT D'EXPLOITATION

# **ALGORITHME DE FORCE BRUTE**

# ALGORITHME BRUTE FORCE

- Brute force est une méthode de résolution de problèmes qui explore toutes les solutions possibles jusqu'à trouver la meilleure.

- **AVANTAGES :**

- Trouver une solution optimale
- Utilisation pour les problèmes de petite taille .

- **INCONVÉNIENTS :**

- Son coût élevé en temps de calcul et en ressources
- Il est inefficace si le nombre de combinaisons possibles est trop grand
- Incapacité à trouver une solution rapide

# TEST DE COMBINAISONS POSSIBLES

Nombre de combinaisons possibles pour 20 actions.

La fonction utilise la fonction **math.comb** qui permet de calculer le nombre de combinaisons possibles de **J** dans un ensemble de **I**.

La fonction effectue une boucle sur tous les 20 actions, et calcule le nombre de combinaisons possibles pour chaque nombre d'actions

```
def compteur_combinaison():  
    for i in range(1, 21):  
        nb_combinations = 0  
        for j in range(1, i+1):  
            nb_combinations += math.comb(i, j)  
        print(f"Nombre de combinaisons pour {i} actions : {nb_combinations}")  
  
compteur_combinaison()
```

```
Nombre de combinaisons pour 1 actions : 1  
Nombre de combinaisons pour 2 actions : 3  
Nombre de combinaisons pour 3 actions : 7  
Nombre de combinaisons pour 4 actions : 15  
Nombre de combinaisons pour 5 actions : 31  
Nombre de combinaisons pour 6 actions : 63  
Nombre de combinaisons pour 7 actions : 127  
Nombre de combinaisons pour 8 actions : 255  
Nombre de combinaisons pour 9 actions : 511  
Nombre de combinaisons pour 10 actions : 1023  
Nombre de combinaisons pour 11 actions : 2047  
Nombre de combinaisons pour 12 actions : 4095  
Nombre de combinaisons pour 13 actions : 8191  
Nombre de combinaisons pour 14 actions : 16383  
Nombre de combinaisons pour 15 actions : 32767  
Nombre de combinaisons pour 16 actions : 65535  
Nombre de combinaisons pour 17 actions : 131071  
Nombre de combinaisons pour 18 actions : 262143  
Nombre de combinaisons pour 19 actions : 524287  
Nombre de combinaisons pour 20 actions : 1048575
```

# DIAGRAMME DE L' ALGORITHME DE FORCE BRUTE

```
def trouver_meilleure_combinaison(actions, budget):  
    """Trouve la meilleure combinaison d'actions qui maximise le profit dans le budget donné"""  
  
    # stocker toutes les combinaisons possibles  
    combinaisons = []  
  
    # fonction combinations pour générer toutes les combinaisons possibles  
    for i in range(1, len(actions) + 1):  
        combinaisons.extend(combinations(actions, i)) #  $O(2^n)$  : boucles imbriquées  
  
    meilleure_combinaison = []  
    meilleur_profit = 0  
    investissement_total = 0  
  
    """ Pour chaque combinaison, on fait le total du prix des actions """  
    for combinaison in combinaisons:  
        total_prix = sum([action.prix for action in combinaison])  
  
        # on vérifie si le total est inférieur ou égal au budget.  
        if total_prix <= budget:  
            total_profit = sum([action.profit() for action in combinaison])  
  
            """  
            On vérifie si le profit total est supérieur au profit total actuel dans la boucle.  
            Si c'est le cas, on met à jour le meilleur profit  
            """  
  
            if total_profit > meilleur_profit:  
                meilleure_combinaison = combinaison  
                meilleur_profit = total_profit  
                investissement_total = total_prix  
  
    return (meilleure_combinaison, investissement_total)
```

1- Tout d'abord, on calcule toutes les combinaisons possibles d'actions parmi les 20 disponibles.

2- Ensuite, on parcourt chaque combinaison d'actions, et pour chacune d'elles, on calcule le coût total et le profit total.

3- On filtre ensuite les combinaisons qui ont un coût total inférieur ou égal à 500.

4- Enfin, on retourne la combinaison qui a le profit total le plus élevé parmi les combinaisons qui respectent le budget de 500.

# RÉSULTATS DE L' ALGORITHME DE FORCE BRUTE

```
liste_achat_actions = [action.nom for action in meilleure_combinaison]
profit = sum(action.profit() for action in meilleure_combinaison)

print("Meilleure combinaison d'actions avec un budget de 500 : ")
print(f"Voici la liste des actions à acheter : {liste_achat_actions}")
print(f"Profit total sur 2 ans : {profit :.2f} euros")
print(f"Investissement total : {investissement_total:.2f} euros")
```

```
Meilleure combinaison d'actions avec un budget de 500 :
Voici la liste des actions à acheter : ['Action-04', 'Action-05']
Profit total sur 2 ans : 99.08 euros
Investissement total : 498.00 euros
```

# **L'ALGORITHME GLOUTON (SOLUTION OPTIMISÉE)**

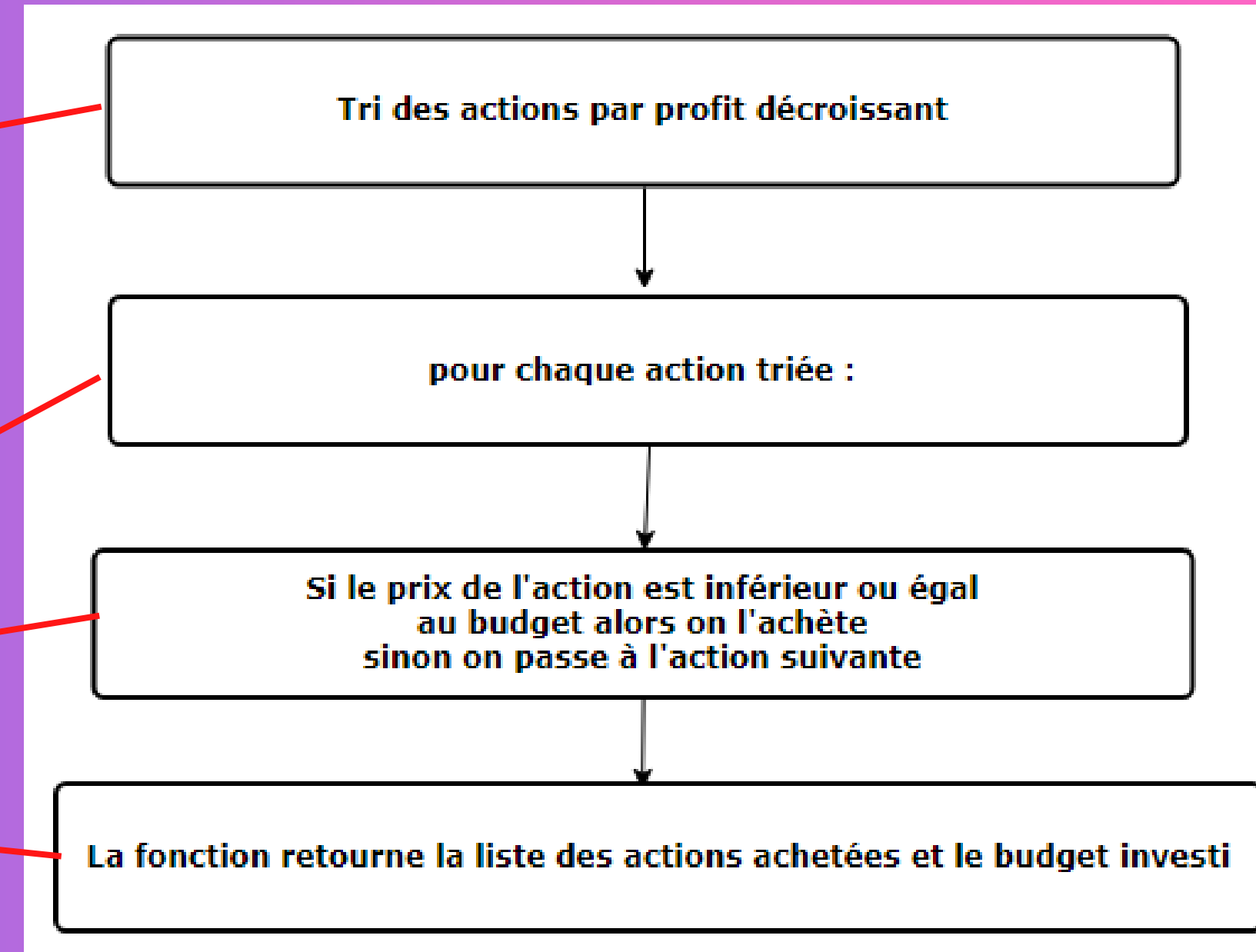
# ALGORITHME GROUTON

- L'algorithme glouton c'est une méthode d'optimisation qui résout un problème en faisant un choix à chaque étape dans l'espoir d'atteindre une solution globale optimale.
- **AVANTAGES :**
  - Il peut donner une solution très rapidement avec un grand nombre de données
- **INCONVÉNIENTS :**
  - il ne donne pas une solution optimale, car les choix ne garantissent pas toujours la meilleure solution.



# DIAGRAMME DE L'ALGORITHME GLOUTON

```
def acheter_actions(budget_max, actions):  
    actions_triees = sorted(actions, reverse=True)  
  
    actions_achetees = []  
    budget_investi = 0  
  
    for action in actions_triees:  
        if action.prix <= budget_max - budget_investi:  
            actions_achetees.append(action)  
            budget_investi += action.prix  
  
    return actions_achetees, budget_investi
```



# RÉSULTATS DE L'ALGORITHME GROUTON

## code

```
# Récupération des actions depuis le fichier CSV
actions1 = recup_action_csv("data/dataset1_Python+P7.csv")
actions2 = recup_action_csv("data/dataset2_Python+P7.csv")

print("-"*40)
print("Résultats pour dataset1_Python+P7.csv :")
print("-"*40)
actions_achetees1, budget_investi1 = acheter_actions(500, actions1)
afficher_resultats(actions_achetees1, budget_investi1)

print("-"*40)
print("Résultats pour dataset2_Python+P7.csv :")
print("-"*40)
actions_achetees2, budget_investi2 = acheter_actions(500, actions2)
afficher_resultats(actions_achetees2, budget_investi2)
```

## résultats

```
-----
Résultats pour dataset1_Python+P7.csv :
-----
```

```
La fonction acheter_actions a pris 0.00117 secondes
```

```
Budget investi : 499.94
```

```
Profit total : 198.51
```

```
Actions achetées :
```

```
['Share-XJMO', 'Share-MTLR', 'Share-KMTG', 'Share-LRBZ',
 'Share-LGWG', 'Share-SKKC', 'Share-QLMK', 'Share-UEZB']
```

```
-----
Résultats pour dataset2_Python+P7.csv :
-----
```

```
La fonction acheter_actions a pris 0.0006 secondes
```

```
Budget investi : 499.98
```

```
Profit total : 197.77
```

```
Actions achetées :
```

```
['Share-PATS', 'Share-JWGF', 'Share-ALIY', 'Share-NDKR',
 'Share-ROOM', 'Share-VCXT', 'Share-YFVZ', 'Share-OCKK']
```

# **L'ANALYSE DES PERFORMANCES DES ALGORITHMES**

# FONCTION POUR LA MESURE DU TEMPS D'EXÉCUTION

```
# Mesurer le temps d'exécution
def performance(fonction):

    """Surveiller le temps d'exécution d'une fonction"""
    def wrapper(*args, **kawrgs):

        # Enregistrer le temps actuel avant l'exécution de la fonction passée en argument
        temps1 = perf_counter()
        # Appeler la fonction passée en argument avec les arguments et les mots-clés fournis
        resultat = fonction(*args, **kawrgs)
        # Enregistrer le temps actuel après l'exécution de la fonction passée en argument
        temps2 = perf_counter()
        print(f"\nLa fonction {fonction.__name__} a pris {round(temps2 - temps1, 5)} secondes")
        return resultat

    return wrapper
```

Cette fonction mesure le temps d'exécution d'une autre fonction.

Elle utilise la fonction **perf\_counter()** du module time pour enregistrer l'heure actuelle au début et à la fin de l'exécution de la fonction : temps1 - temps2

Ensuite, elle calcule la différence entre les deux temps pour obtenir le temps écoulé pour l'exécution de la fonction.

# COMPARAISON DE MESURE DU TEMPS D'EXÉCUTION POUR LA BRUTE FORCE ET GLOUTON

20 actions : brute force

```
La fonction trouver_meilleure_combinaison a pris 3.69803 secondes
```

```
Voici la liste des actions à acheter : ['Action-04', 'Action-05',  
Profit total sur 2 ans : 99.08 euros  
Investissement total : 498.00 euros  
Utilisation de mémoire courante : 3.06161 MB  
Utilisation de mémoire maximale : 65.522034 MB
```

dataset 1 (1000 actions) et  
dataset 2 (1000 actions):  
algorithmme glouton

```
-----  
Résultats pour dataset1_Python+P7.csv :  
-----
```

```
La fonction acheter_actions a pris 0.00116 secondes
```

```
Budget investi : 499.94  
Profit total : 198.51  
Actions achetées :  
['Share-XJMO', 'Share-MTLR', 'Share-KMTG', 'Share-LRBZ',  
 'Share-LGWG', 'Share-SKKC', 'Share-QLMK', 'Share-UEZB',
```

```
-----  
Résultats pour dataset2_Python+P7.csv :  
-----
```

```
La fonction acheter_actions a pris 0.00059 secondes
```

```
Budget investi : 499.94  
Profit total : 197.77  
Actions achetées :  
['Share-PATS', 'Share-JWGF', 'Share-ALIY', 'Share-NDKR',  
 'Share-ROOM', 'Share-VCXT', 'Share-YFVZ', 'Share-OCKK',
```

```
Utilisation de mémoire courante : 0.402511 MB  
Utilisation de mémoire maximale : 0.433709 MB
```

# MESURE DE LA MÉMOIRE

Utilisation du module **tracemalloc** : suivre le temps d'exécution de la mémoire par le programme.

La méthode **get\_traced\_memory()** : récupérer la quantité de mémoire utilisée par le programme

```
def main():
    tracemalloc.start()

    # Récupération des données à partir du fichier csv
    actions = recup_action_csv("data/action.csv")

    # Recherche de la combinaison d'actions qui maximise le profit dans le budget de 500 euros
    meilleure_combinaison, investissement_total = trouver_meilleure_combinaison(actions, 500) #  $O(2^n)$  /  $O(n^2)$ 

    liste_achat_actions = [action.nom for action in meilleure_combinaison]
    profit = sum(action.profit() for action in meilleure_combinaison)

    print("Meilleure combinaison d'actions avec un budget de 500 : ")
    print(f"Voici la liste des actions à acheter : {liste_achat_actions}")
    print(f"Profit total sur 2 ans : {profit :.2f} euros")
    print(f"Investissement total : {investissement_total:.2f} euros")

    # Obtenez la quantité de mémoire utilisée pendant l'exécution de la fonction
    memoire_courante, memoire_max = tracemalloc.get_traced_memory()
    print(f"Utilisation de mémoire courante : {memoire_courante / 10**6} MB")
    print(f"Utilisation de mémoire maximale : {memoire_max / 10**6} MB")

    tracemalloc.stop()
```

# MESURE DE LA MÉMOIRE ENTRE LA BRUTE FORCE ET GLOUTON

## 20 actions : brute force

```
La fonction trouver_meilleure_combinaison a pris 3.69803 secondes
Meilleure combinaison d'actions avec un budget de 500 :
Voici la liste des actions à acheter : ['Action-04', 'Action-05',
Profit total sur 2 ans : 99.08 euros
```

```
Utilisation de mémoire courante : 3.06161 MB
Utilisation de mémoire maximale : 65.522034 MB
```

## dataset 1 (1000 actions) et dataset 2 (1000 actions): algorithme glouton

```
-----
Résultats pour dataset1_Python+P7.csv :
-----
```

```
La fonction acheter_actions a pris 0.00116 secondes
Budget investi : 499.94
Profit total : 198.51
```

```
Actions achetées :
```

```
['Share-XJMO', 'Share-MTLR', 'Share-KMTG', 'Share-LRBZ',
'Share-LGWG', 'Share-SKKC', 'Share-QLMK', 'Share-UEZB',
```

```
-----
Résultats pour dataset2_Python+P7.csv :
-----
```

```
La fonction acheter_actions a pris 0.00059 secondes
Budget investi : 499.98
Profit total : 197.77
```

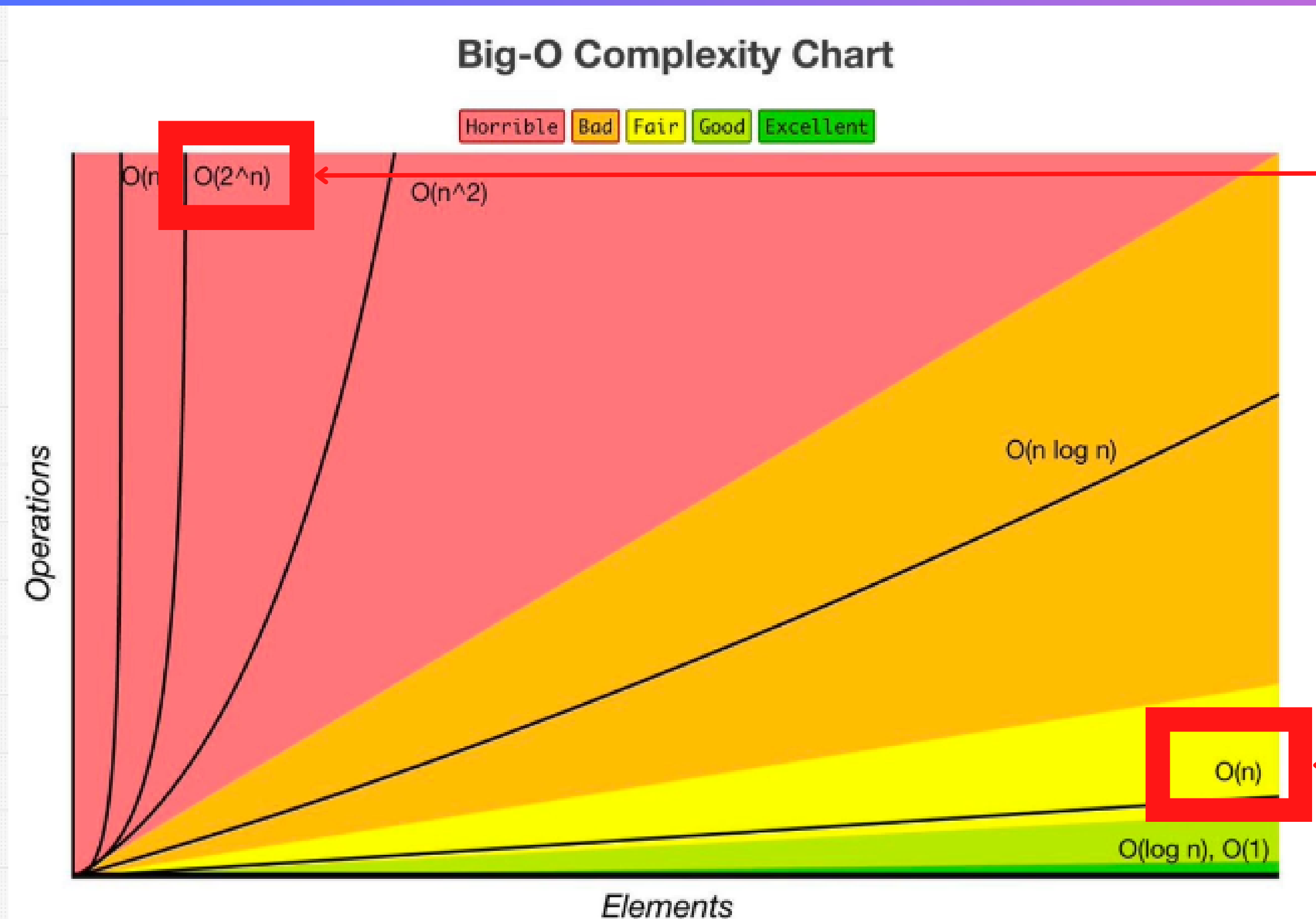
```
Actions achetées :
```

```
['Share-PATS', 'Share-JWGF', 'Share-ALIY', 'Share-NDKR',
'Share-ROOM', 'Share-VCXT', 'Share-YFVZ', 'Share-OCKK',
```

```
Utilisation de mémoire courante : 0.402511 MB
Utilisation de mémoire maximale : 0.433709 MB
```

# NOTATION BIG O

Définition : C'est une méthode de mesure de la performance et de la rapidité d'un algorithme



**Brute force** : elle utilise une boucle for imbriquée dans une autre boucle for.

**Algorithme glouton** : elle utilise une boucle for pour parcourir les actions triées par profit



**COMPARAISON**  
**ENTRE**  
**LES RÉSULTATS DE L'ALGORITHME**  
**ET**  
**LES CHOIX DE SIENNA**

# DATASET 1

	sienna	algorithmme glouton
action(s) acheté(es)	1	25
coût total	498.76	499.94
profit total	196.61	198.51

# DATASET 2

	sienna	algorithme glouton
action(s) acheté(es)	18	22
coût total	489.24	499.98
profit total	193.78	197.77

# RAPPORT D'EXPLORATION

## PRIX OU PROFIT À ZÉRO OU INFÉRIEUR À ZÉRO

dataset1 :

45 actions erronées sur 1000 actions

dataset2 :

459 actions erronées sur 1000 actions

## DONNÉES EXPLOITABLES

dataset1 :

955 actions exploitables

dataset2 :

541 actions exploitables