Table des matières

1 Présentation du projet

L'étude de la complexité algorithmique tient une place prépondérante lorsqu'on s'intéresse à des programmes devant traité un grand nombre de données. En effet, entre deux programmes ayant la même finalité, il se peut que la différence de temps d'exécution soit comptée en jours. C'est pourquoi, il est important de choisir un algorithme optimal pour chaque programme. Mais bien que la complexité de certains programmes soit bien connue (telle que les tris de tableau), certaines complexités sont plus compliquées à prédire. Le but de ce stage fut de trouver une façon de connaître la complexité de tout algorithme en se basant seulement sur leur temps d'exécution.

1.1 Sujet

Comme dit dans la courte introduction, le but de ce stage fut de trouver la complexité des algorithmes en se basant sur leur temps d'exécution. Le projet s'est donc décomposé en deux phases. Durant la première, le but fut de retrouver la complexité d'algorithmes bien connus (tri à bulles par exemple) afin de vérifier que la méthode utilisée donnait des résultats corrects. Ensuite, le programme est testé sur des codes étudiants posté sur caseine [1].

2 Principes

2.0.1 Explications

Le problème est de savoir si on peut déterminer automatiquement la complexité temporelle au pire cas d'un algorithme en disposant de son code source et de la possibilité d'effectuer des tests numériques dans un temps limité.
Dans ce travail on va se restreindre à 6 classes de complexité :

$$log(n), n, nlog(n), n^2, n^2log(n), n^3$$

On propose d'exploiter les temps d'exécutions sur des instances de différentes tailles et de procéder par régression linéaire pour identifier la complexité qui semble refléter au mieux l'exécution de l'algorithme.
Deux approches sont évaluées dans la suite de ce document.
La première consiste à faire une unique régression $H_\theta(n)$ multi-variables prenant la forme suivante :

$$H_\theta(n) = \theta_0 + \theta_1 log(n) + \theta_2 n + \theta_3 nlog(n) + \theta_4 n^2 + \theta_5 n^2 log(n) + \theta_6 n^3$$

On examine ensuite les coefficients θ_i de la régression pour tenter d'isoler le ou les termes le plus significatifs qui pourraient indiquer la complexité réelle de l'algorithme.
La deuxième consiste à effectuer une régression par classe de complexité donc six régressions distinctes :

$$H^1_\theta(n) = \theta_0 + \theta_1 log(n)$$
$$H^2_\theta(n) = \theta_0 + \theta_1 n$$
$$H^4_\theta(n) = \theta_0 + \theta_1 n^2$$
$$H^6_\theta(n) = \theta_0 + \theta_1 n^3$$

On examine ensuite l'erreur obtenue (la somme des carrés des écarts aux points connus) pour retenir comme complexité réelle de l'algorithme la regression H^k minimisant cette erreur car on propose de retenir comme complexité la régression donnant l'erreur minimum puisque l'erreur quantifie l'adéquation de la fonction aux temps d'exécution observés.

1. http ://caseine.org/

Dans cette deuxième méthode, on se passe volontairement de $O(nlog(n))$ et $O(n^2log(n))$ car leur temps d'exécution sont très proche de $O(n)$ (resp. $O(n^2)$) ce qui les rends très difficile a distinguer avec cette méthode.

2.0.2 Protocole

- Dans un premier temps on éxecute l'algorithme pour toutes les tailles de tableau en mesurant le temps d'éxecution à chaque fois

- Une fois le temps mesuré, on construit le tableau des features.

$log(n)$	n	$nlog(n)$	n^2	$n^2log(n)$	n^3	Temps CPU
9.21	10000	92103.40	$1.0 * 10^8$	$9.0 * 10^8$	$1.0 * 10^{12}$	$56ms$
9.90	20000	198068.75	$4.0 * 10^8$	$4.0 * 10^9$	$8.0 * 10^{12}$	$187ms$
10.31	30000	309268.58	$9.0 * 10^8$	$9.0 * 10^9$	$2.7 * 10^{13}$	$491ms$
10.60	40000	423865.39	$1.6 * 10^8$	$2.0 * 10^{10}$	$6.4 * 10^{13}$	$884ms$
10.82	50000	540988.91	$2.5 * 10^9$	$3.0 * 10^{10}$	$1.3 * 10^{14}$	$1482ms$
11.00	60000	660126.00	$3.6 * 10^9$	$4.0 * 10^{10}$	$2.2 * 10^{14}$	$1612ms$
11.16	70000	780937.54	$4.9 * 10^9$	$5.0 * 10^{10}$	$3.4 * 10^{14}$	$2139ms$
11.29	80000	903182.55	$6.4 * 10^9$	$7 * 10^{10}$	$5.1 * 10^{14}$	$2962ms$
11.41	90000	1026680.85	$8.1 * 10^9$	$9.0 * 10^{10}$	$7.3 * 10^{14}$	$3708ms$
11.51	100000	1151292.55	$1.0 * 10^{10}$	$1.0 * 10^{11}$	$1.0 * 10^{15}$	$4608ms$

FIGURE 1 – Exemple de tableau utilisé pour la régression

- On réalise la régression linéaire $H_\theta(n)$.

- Dans un premier temps, avec la régression $H_\theta(n)$,on va s'interesser à ses coefficients θ_i afin de trouver le terme permettant de conclure sur la complexité. Si ces termes ne nous permettent pas de conclure (i.e fausse complexité) alors, on réalise la deuxième approche.

- Dans la deuxième approche, on effectue une régression par classe de complexité puis on calcule l'erreur obtenue pour chaqu'une d'entre elle. On retient ensuite le H^k ayant l'erreur minimal et permettant de conclure sur la complexité.

- On trace le graphique contenant toutes les régressions linéaire ainsi que l'éxecution de l'algorithme.

- On extrait un tableau contenant l'erreur résiduelle pour chaque $H^i_\theta(n)$ et le temps d'éxecution total.

2.1 Proof of concept

Dans cette partie, on s'intresse à des algorithmes bien connu afin de déterminer si notre méthode fonctionne ou non.

2.1.1 Tri à bulles simple $O(n^2)$

Tailles de tableau utilisées : 2500, 5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000
Complexité théorique : $O(n^2)$

Lors qu'on exécute l'algorithme avec des entrées differentes et que l'on mesure le temps d'exécution, on obtient le graphique suivant :

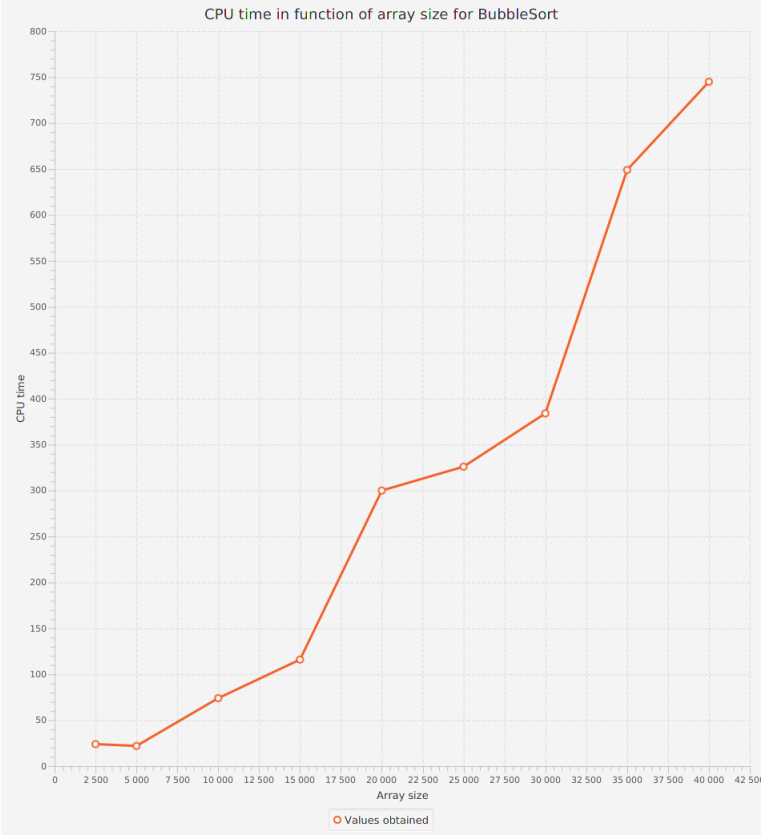

FIGURE 2 – Le temps d'exécution de l'algorithme en ms en fonction de la taille du tableau

Le programme procède ensuite à la régression.
On obtient la régression suivant :
$\mathbf{H(n) = 257498.66 - 35867.89log(n) + 54.14n - 5.30nlog(n) + (7.10 * 10^{-4})n^2}$
$\mathbf{-(5.52 * 10^{-5})n^2log(n) + (1.13 * 10^{-10})n^3}$
Or, dans ce cas, si on choisit de prendre le plus grand θ afin de déterminer la complexité, ici on obtient une complexité en $O(n)$ ce qui est incorrecte, on doit donc utiliser la deuxième méthode qui consiste a réaliser une régression linéaire par classe de complexité $(H_\theta^i(n))$ et de calculer l'erreur.

On obtient alors les résultats suivants :

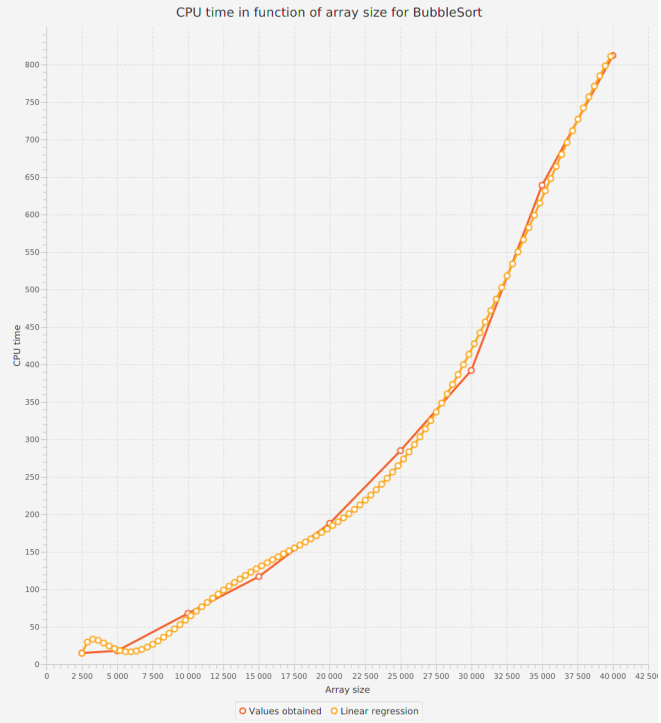

FIGURE 3 – Régression linéaire en ms en fonction de la taille des données (n)

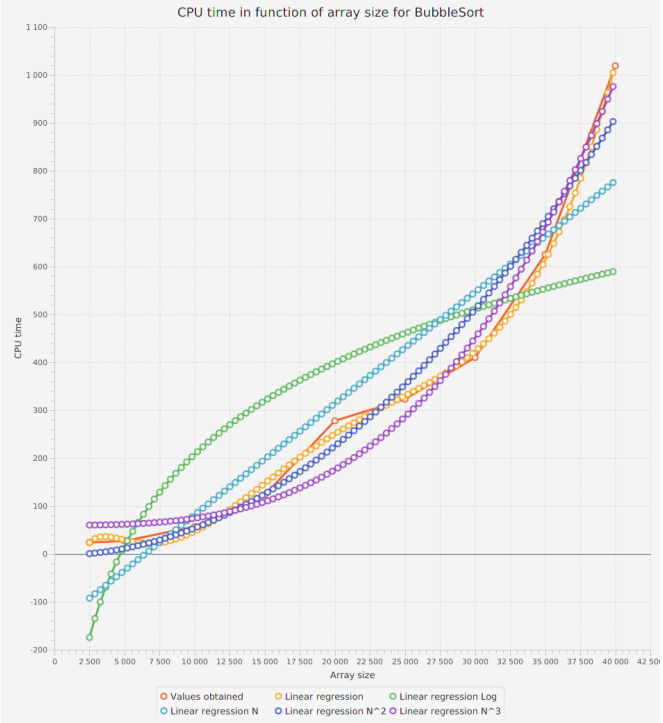

FIGURE 4 – Toutes les régressions linéaire en ms en fonction de la taille des données (n)

On obtient les erreurs suivantes :

Régression linéaire	$H_\theta(n)$	$H_\theta^1(n)$	$H_\theta^2(n)$	$H_\theta^4(n)$	$H_\theta^6(n)$
Erreur résiduelle :	0.06	29280.90	9225.85	**146.96**	1772.99
Indicateur sur les erreurs :	0	19824%	6177%	**0**%	1106%

FIGURE 5 – Tableau récapitulatif des erreurs résiduelles

L'indicateur de la seconde ligne est calculer comme ceci :

$$\frac{H_\theta^k - H_\theta^{min}}{H_\theta^{min}}$$

Où H_θ^{min} correspond au $H_\theta^k(n)$ ayant la valeur d'erreur la plus petite.
De plus, ici l'indicateur pour $H_\theta(n)$ est 0 car on ne compare que les $H_\theta^k(n)$ entre eux, on peut donc s'en

passer pour les futurs algorithmes a tester.

L'erreur résiduelle la plus faible étant pour la régression de la forme $\mathbf{H_\theta^4(n) = \theta_0 + \theta_1 n^2}$, on obtient bien une compléxité en $\mathbf{O(n^2)}$ comme voulu.

On cherche ensuite à déterminer la véracité (i.e le programme donne la bonne complexité) de nos résultats, ainsi que le temps moyen d'exécution.

% de bon résultats	Temps d'exécution moyen
99%	$2.5s.$

FIGURE 6 – Taux de réussite et temps d'exécution moyen du programme

2.1.2 Tri à bulles parasité $O(k_1 n + k_2 nlog(n) + n^2)$

De la même façon que l'exemple précédent, sauf que cette fois, on parasite le tri à bulles en rajoutant un tri fusion au milieu (sur une copie du tableau) et des parcours de tableau. Ceci a pour but d'avoir une compléxité du type $O(k_1 n + k_2 nlog(n) + n^2)$ afin de voir si la méthode utilisée donne bien une compléxité en $O(n^2)$.
Ceci à pour but de montrer la robustesse de l'algorithme.
On applique ensuite la même méthode pour determiné la complexité.
Tailles de tableau utilisées : 2500, 5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000
Complexité théorique : $O(n^2)$

On obtient les graphiques suivants :

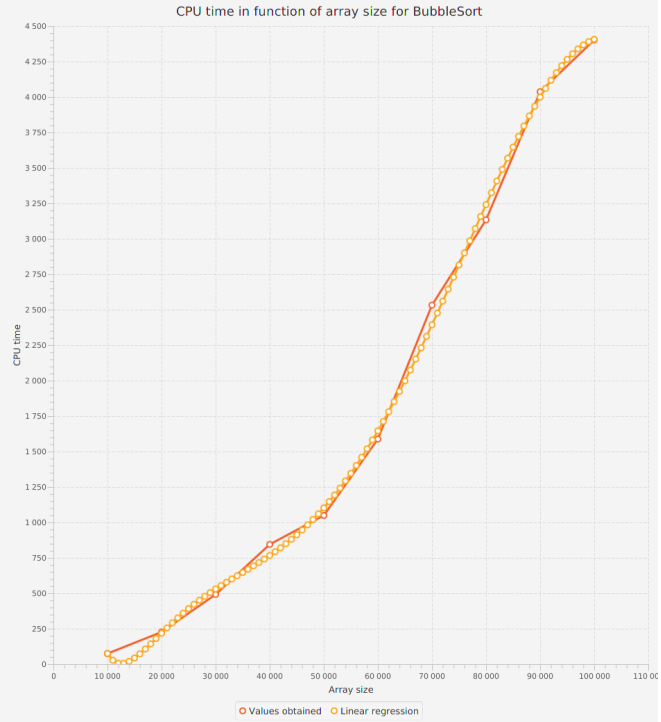

FIGURE 7 – Régression linéaire en ms en fonction de la taille des données (n)
Avec $k1 = 10$ et $k2 = 10$

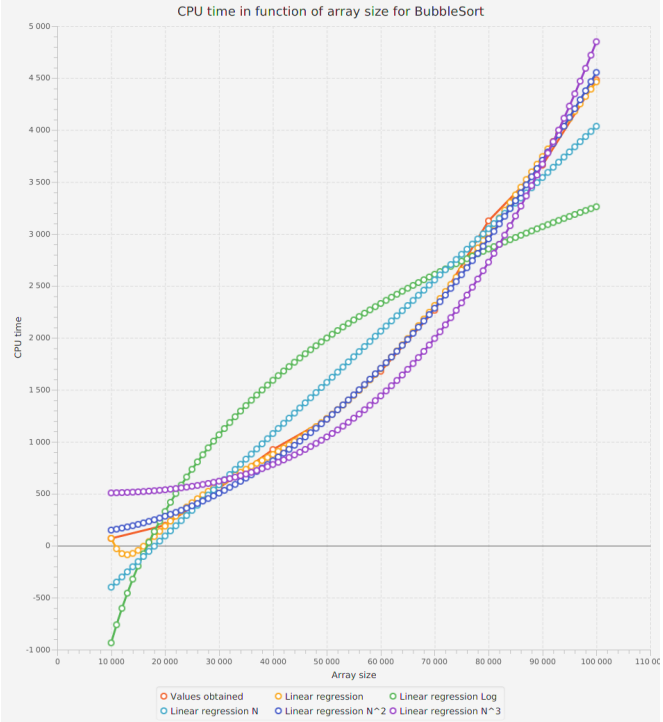

FIGURE 8 – Toutes les régressions linéaire en ms en fonction de la taille des données (n)
Avec $k1 = 10$ et $k2 = 10$

Et les erreurs résiduelles suivantes :

k_1	k_2	$H_\theta^1(n)$	$H_\theta^2(n)$	$H_\theta^4(n)$	$H_\theta^6(n)$
10	10	27491%	6375%	**0%**	4444%
10	100	6438%	2158%	**0%**	30033%
100	10	$1.03 * 10^8$%	$2.83 * 10^7$%	**0%**	$1.21 * 10^7$%
100	100	4406%	735%	**0%**	1080%
1000	10	$6.77 * 10^5$%	$1.86 * 10^5$%	**0%**	66379%
1000	100	40%	78%	**0%**	496%

FIGURE 9 – Indicateur sur les erreurs résiduelles

L'erreur résiduelle la plus faible étant pour la régression de la forme $\mathbf{H_\theta^4(n) = \theta_0 + \theta_1 n^2}$, on obtient bien une compléxité en $\mathbf{O(n^2)}$ comme voulu. Cependant, on peut voir que si l'on parasite énormement le code de base, on n'obtient difficilement plus de 10% de bons résultats ainsi que des temps d'exécution bien trop long.

Une autre façon de tester la robustesse de notre programme est de changer les tailles d'entrer afin de regarder a partir de quelle taille maximal on obtient les bons résultats. Ceci à pour but de diminuer au maximum le temps d'exécution de l'algorithme.

k_1	k_2	Temps exécution moyen	% de bon résultats
10	10	$2.34s.$	96%
10	100	$3.78s.$	81%
100	10	$2.82s.$	94%
100	100	$4.13s.$	76%
1000	10	$2.91s.$	92%
1000	100	$4.256s.$	70%

FIGURE 10 – Taux de réussite et temps d'exécution moyen du programme

k_1	k_2	Temps exécution moyen	% de bon résultats
10	1000	$15.98s.$	15%
100	1000	$19.44s.$	10%
1000	1000	$18.88s.$	8%

FIGURE 11 – Taux de réussite et temps d'exécution moyen du programme

Taille maximal	Temps d'exécution total	% de bon résultats
40000	$2.5s.$	78%
25000	$1.1s.$	90%
50000	$4.8s.$	66%
100000	$28.2s.$	88%

FIGURE 12 – Taux de réussite et temps d'exécution du programme en fonction de la taille maximal des entrées

2.1.3 Power plant $O(n)$

Tailles de tableau utilisées : 10000000, 15000000, 20000000, 25000000, 30000000, 35000000, 40000000, 45000000, 50000000
Complexité théorique : $O(n)$

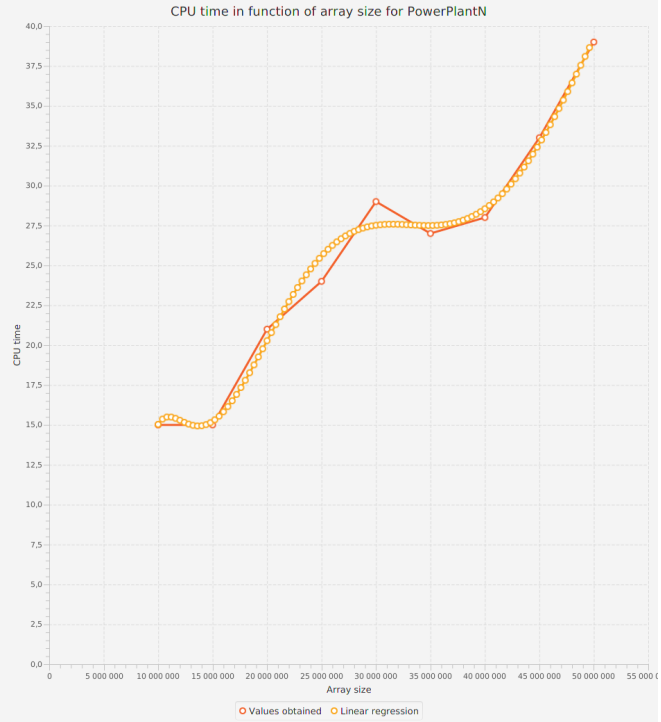

FIGURE 13 – Régression linéaire en ms en fonction de la taille des données (n)

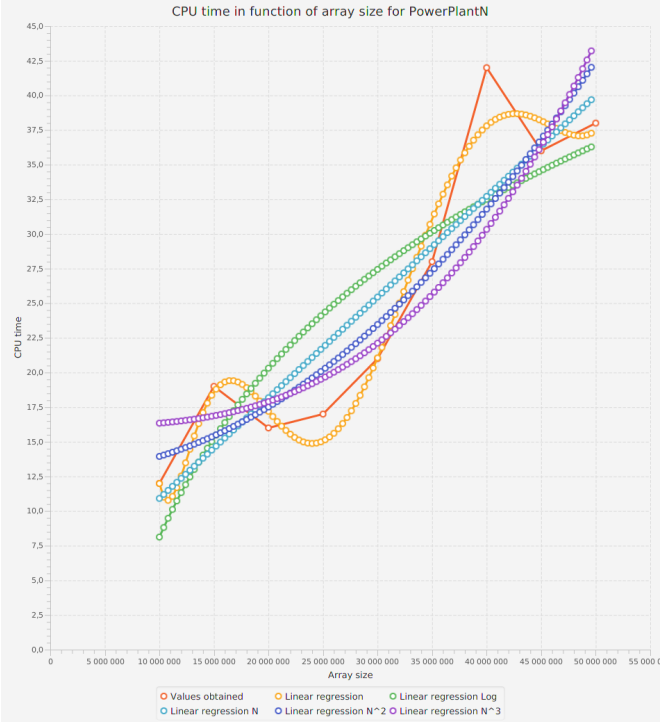

FIGURE 14 – Toutes les régressions linéaire en ms en fonction de la taille des données (n)

Les erreurs résiduelles suivantes :

$H_\theta^1(n)$	$H_\theta^2(n)$	$H_\theta^4(n)$	$H_\theta^6(n)$
1171%	**0**%	219%	1498%

FIGURE 15 – Indicateur sur les erreurs résiduelles

% de bon résultats	Temps d'éxécution moyen
66%	$0.2s.$

FIGURE 16 – Taux de réussite et temps d'exécution moyen du programme

L'erreur résiduelle la plus faible étant pour la régression de la forme $\mathbf{H_\theta^2(n) = \theta_0 + \theta_1 n}$, on obtient bien une compléxité en $\mathbf{O(n)}$ comme voulu.

2.1.4 Produit matriciel $O(n^3)$

Tailles de tableau utilisées : 300, 350, 400, 450, 500, 550, 600, 650, 700
Complexité théorique : $O(n^3)$

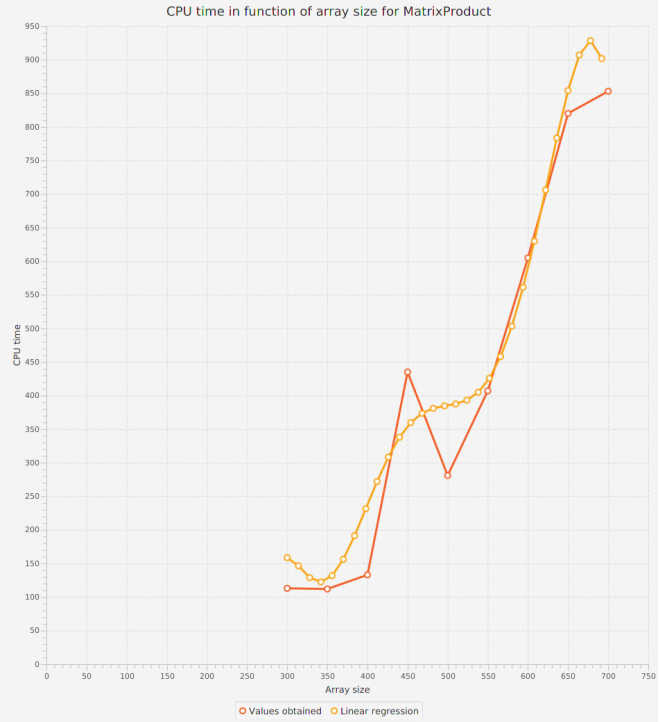

FIGURE 17 – Régressions linéaire en ms en fonction de la taille des données (n)

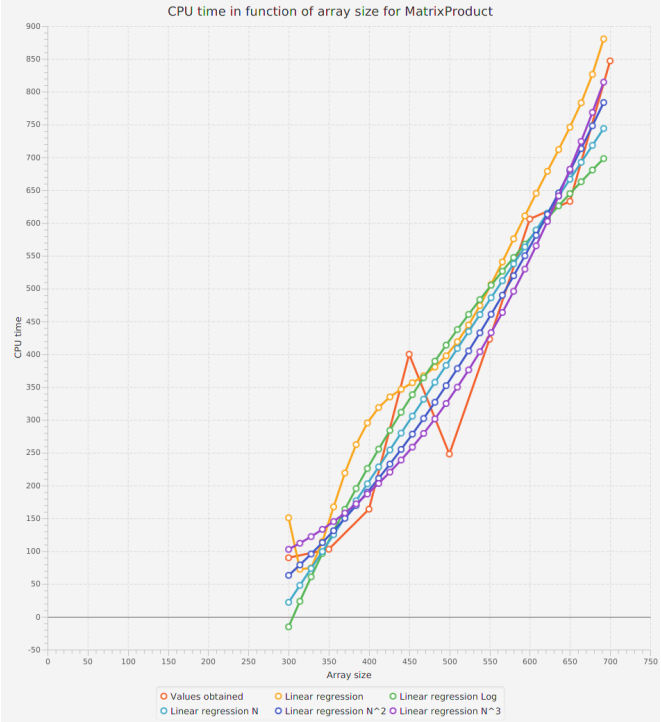

FIGURE 18 – Toutes les régressions linéaire en ms en fonction de la taille des données (n)

$H_\theta^1(n)$	$H_\theta^2(n)$	$H_\theta^4(n)$	$H_\theta^6(n)$
137%	66.7%	19.6%	**0**%

FIGURE 19 – Indicateur sur les erreurs résiduelles

% de bon résultats	Temps moyen d'exécution
71%	$3.7s.$

FIGURE 20 – Taux de réussite et temps d'exécution moyen du programme

L'erreur résiduelle la plus faible étant pour la régression de la forme $\mathbf{H_\theta^6(n) = \theta_0 + \theta_1 n^3}$, on obtient bien une compléxité en $\mathbf{O(n^3)}$ comme voulu.

2.1.5 Conclusion

Suite à ces différentes expérimentations sur des algorithmes, les résultats permettant la conclusion suivante. Pour les programmes à complexité "simple" (i.e $O(n), O(n^2), O(n^3)$) le programme donne un taux de bonnes réponses satisfaisantes. Cependant, si le l'algorithme testé possède une somme de différentes complexité, le taux de bonne réponse chute drastiquement (Moins de 10% dans certains cas).

2.2 Résultats expérimentaux sur code étudiant

3 Machine learning

Suite au calcul de la somme des carrés de l'erreur résiduelle, on obtient le bon résultat avec une complexité en $O(n^2)$ ici pour le tri à bulles. Cependant, il reste un petit problème a résoudre, comment distinguer une complexité en $O(n)$ et une en $O(nlog(n))$ ainsi que une complexité en $O(n^2)$ et une en $O(n^2log(n))$ car celles-ci sont très proche l'une de l'autre ce qui complique le calcul de l'erreur résiduelle.

3.1 Etude du sujet

Intro

3.2 Problème posé

Bla

3.3 Solution proposée

Bla

Transition

3.4 Réalisation