TRAVAIL PRATIQUE 1
INF4705 – Automne 2016

POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

Le 16 octobre 2016

Adrien Doumergue (1868995) & Robin Royer (1860715)

Introduction

Dans le cadre de ce laboratoire, nous allons effectuer l'analyse empirique et hybride de deux algorithmes de tri bien connus: **Merge Sort & Bucket Sort**, ces résultats ont été générés sur un poste fixe de Polytechnique Montréal (4 cores sur 1 processeur intel i5-4570S cadencé à 2.90 GHz et avec 16336 Mo de mémoire vive et version 24 de fedora).
Les algorithmes sont implémentés en Python version 3.

Analyse des jeux de données:

Nous avons considéré dans ce travaux pratique 5 types de jeux de données:

- Jeu de données avec 1000 nombres
- Jeu de données avec 5000 nombres
- Jeu de données avec 10000 nombres
- Jeu de données avec 50000 nombres
- Jeu de données avec 100000 nombres

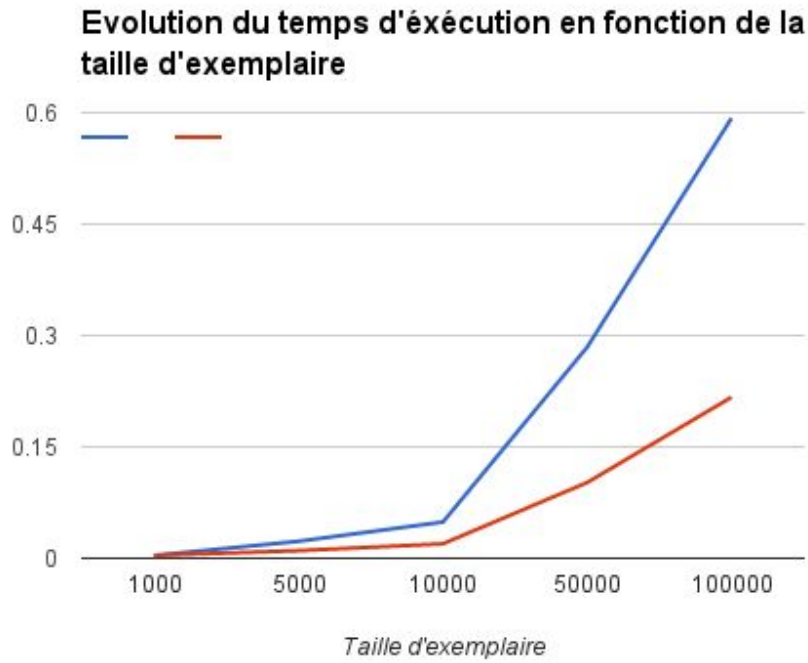
Chaque type de jeu de données avec 3 série de 10 exemplaires différents. Nous avons donc 150 exemplaires différents.

Résultats globaux:

Jeu de données	**Série**	**Bucket sort(s)* ****	**Merge sort(s)****
1000	1	0.00381	0.00385
1000	2	0.00417	0.00372
1000	3	0.00420	0.00372
5000	1	0.01025	0.02287
5000	2	0.01199	0.02244
5000	3	0.01215	0.02227
10000	1	0.01949	0.04878
10000	2	0.02308	0.04770
10000	3	0.02486	0.04756

50000	1	0.10208	0.28448
50000	2	0.11729	0.27313
50000	3	0.11925	0.27681
100000	1	0.21689	0.59270
100000	2	0.23665	0.57557
100000	3	0.23929	0.57522

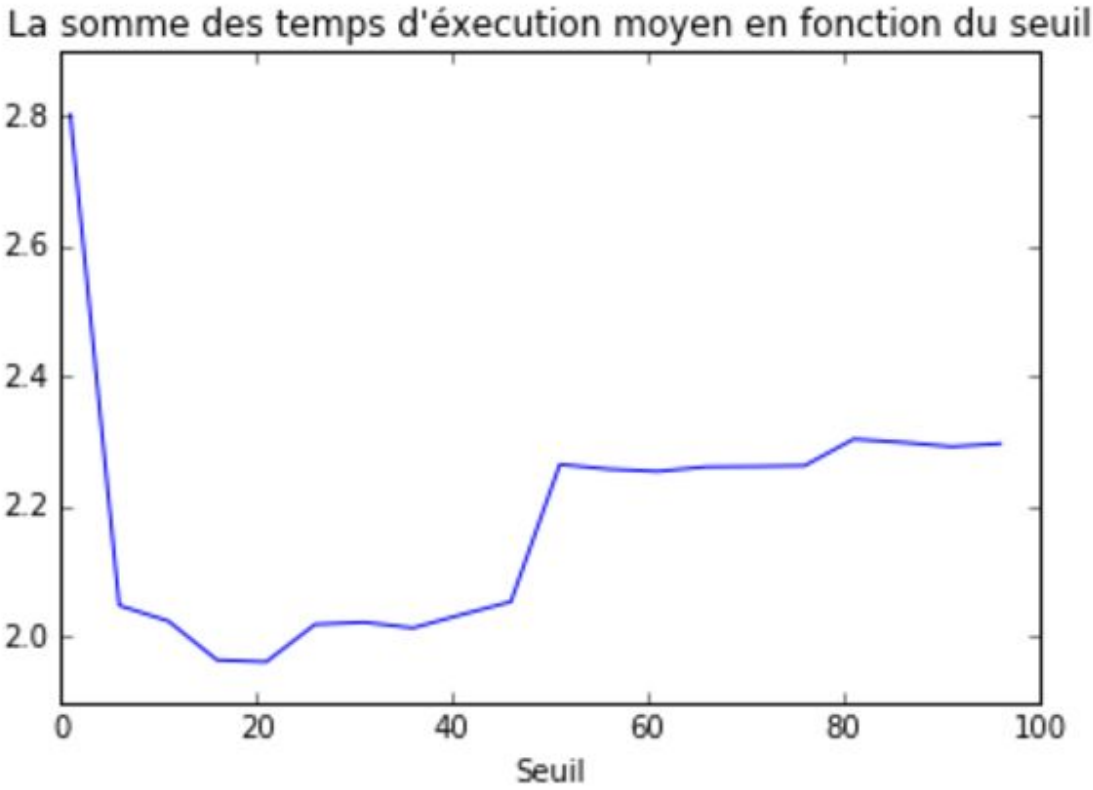
*Ici le nombre de bucket est égale à la longueur de la séquence.
**Ici le seuil de récursivité est égale à 1.



Etude de Merge Sort

a) Seuil

Pour déterminer le seuil optimal pour Merge, nous avons calculé le temps moyen en fonction du seuil pour chaque taille et chaque série et nous avons ensuite cherché le seuil qui minimise la somme (sur la taille et les séries) des temps d'exécution moyen.



Nous avons testé pour les seuils de 1 à 100 avec un pas de 5 et on réalise que le seuil optimal pour la fonction **Merge** est **21**.

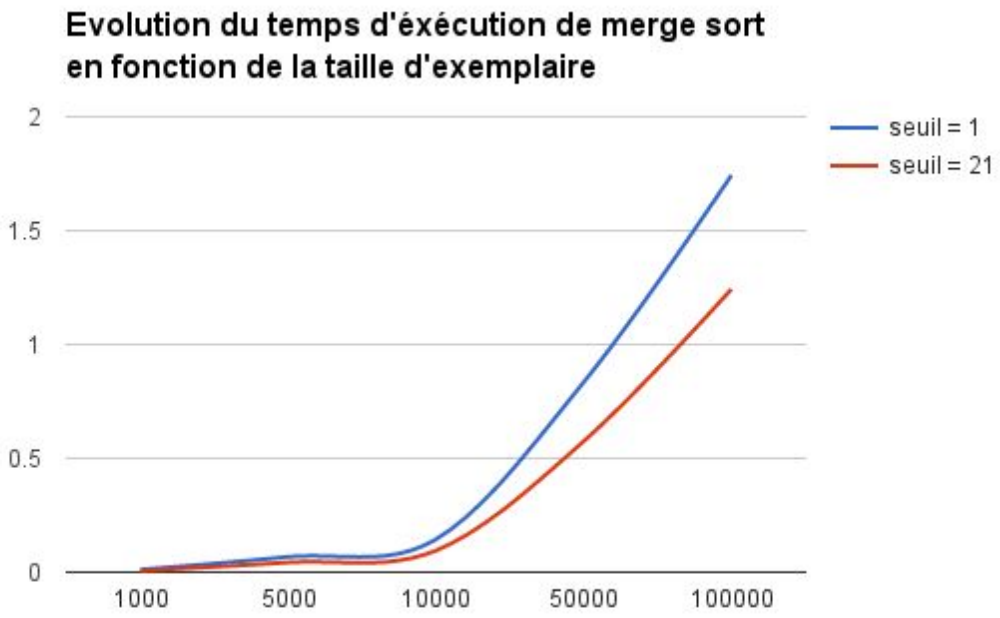
b) Résultats globaux

Nous allons donc prendre en compte le seuil expérimental et optimal pour l'algorithme Merge sort:

jeu de données	série	Merge sort(s)*
1000	1	0.00213
1000	2	0.00171
1000	3	0.00244
5000	1	0.01439
5000	2	0.01183
5000	3	0.01669
10000	1	0.03177
10000	2	0.02668

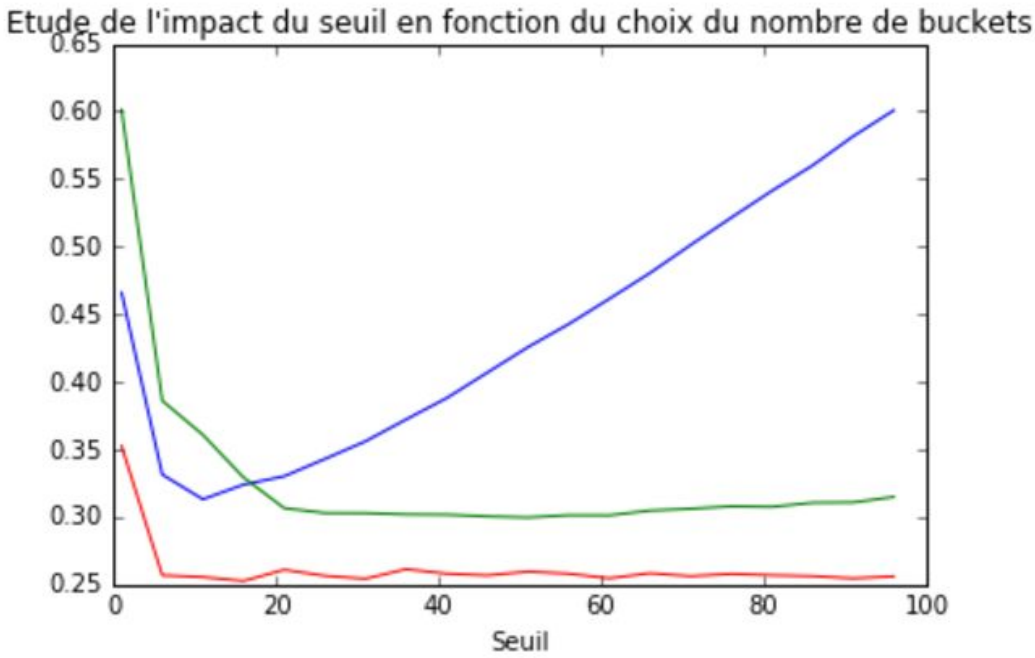
10000	3	0.03628
50000	1	0.19397
50000	2	0.17811
50000	3	0.20287
100000	1	0.42112
100000	2	0.38465
100000	3	0.43744

Nous remarquons alors que le choix d'un seuil efficace pour nos exemplaires est significatif et influe sur la constante: gain de 24% du temps d'exécution, pour l'exemplaire a 100 000 nombres.



Etude de Bucket Sort

a) Seuil et nombre de buckets



L'exécution de l'algorithme *Bucket Sort* dépend de 2 paramètres:

- Le choix du seuil de récursivité.
- Le nombre de buckets pour chaque appel.

Nous étudions alors l'impact du seuil en considérant 3 différents choix de bucket:

- Pour la courbe verte: $2 + \sqrt{taille\ de\ l'exemplaire}$
- Pour la courbe bleu: $2 + taille\ de\ l'exemplaire / Seuil$
- Pour la courbe rouge: *taille de l'exemplaire*

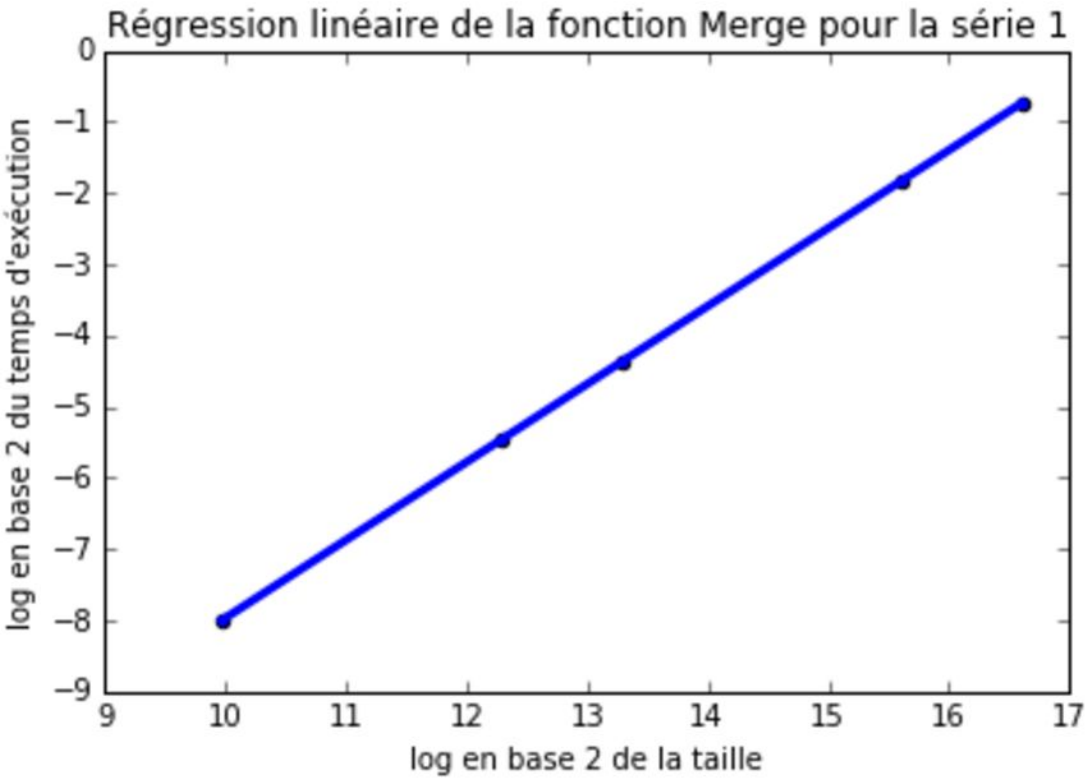
Le choix de taille de l'exemplaire comme nombre de bucket semble être le plus efficace et la meilleur valeur est atteinte pour un **seuil** valant **6**.

Cependant, la courbe bleu est bien représentative de l'étude d'un choix de seuil, séparant l'espace en 2 autour d'un minimum global:

- La partie de gauche est coûteuse car avec un petit seuil, nous passons beaucoup de temps dans les appels récursifs
- La partie de droite quant à elle est coûteuse car avec des buckets de tailles trop importantes, on perd le bénéfice de la récursivité et la complexité dépend de celle de notre algorithme de tri par insertion.

Test de puissance

a) Algorithme Merge Sort

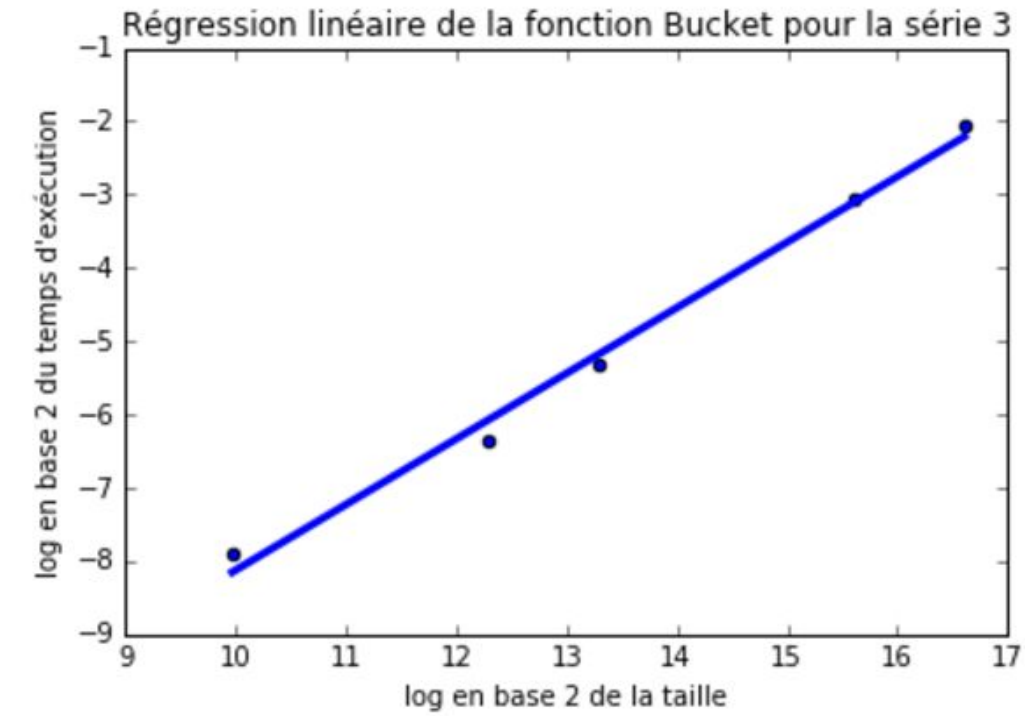


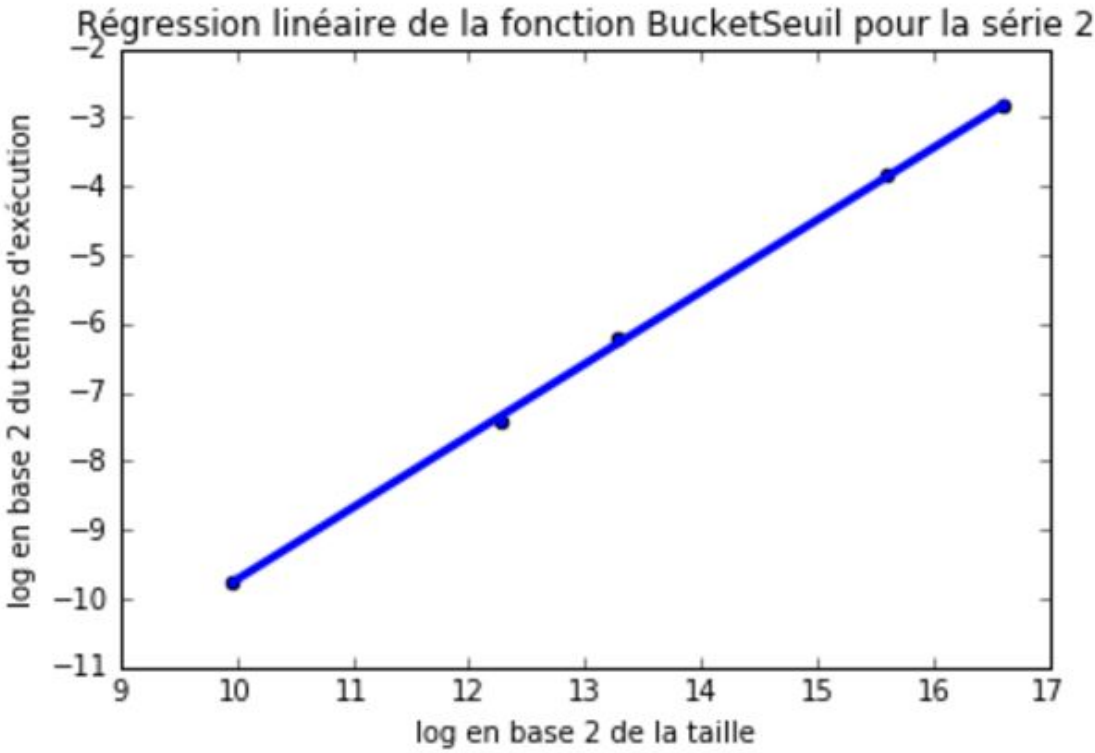
Série/Merge	**Pente**	**Constante**	$y = 2^b * x^m$
1	1.094093	-18.90591	$y = 2.03 \cdot 10^{-6} \cdot x^{1.09}$
2	1.093176	-18.93574	$y = 1.99 \cdot 10^{-6} \cdot x^{1.09}$
3	1.095250	-18.96354	$y = 1.96 \cdot 10^{-6} \cdot x^{1.09}$

Série/MergeSeuil	**Pente**	**Constante**	$y = 2^b * x^m$
1	1.144866	-20.23119	$y = 8.12 \cdot 10^{-7} \cdot x^{1.14}$
2	1.177399	-20.89359	$y = 5.13 \cdot 10^{-7} \cdot x^{1.18}$
3	1.119200	-19.73933	$y = 1.14 \cdot 10^{-6} \cdot x^{1.12}$

On remarque que le MergeSeuil augmente légèrement l'ordre mais réduit considérablement la constante multiplicative. On retrouvera bien des résultats pour MergeSeuil plus bas que ceux de Merge pour des tailles raisonnables mais à l'infini, cette régression linéaire n'est pas bonne. Ce facteur légèrement supérieur à 1 dans les 3 séries nous fait penser que l'analyse théorique fera apparaître un **O(n.log(n)).**

b) Algorithme Bucket Sort





Série/Bucket	**Pente**	**Constante**	$y = 2^b * x^m$
1	0.894606	-17.28733	$y = 6.25 \cdot 10^{-6} \cdot x^{0.89}$
2	0.894144	-17.09638	$y = 7.13 \cdot 10^{-6} \cdot x^{0.89}$
3	0.894462	-17.06532	$y = 7.29 \cdot 10^{-6} \cdot x^{0.89}$

Série/BucketSeuil	**Pente**	**Constante**	$y = 2^b * x^m$
1	1.060567	-20.28475	$y = 7.82 \cdot 10^{-7} \cdot x^{1.06}$
2	1.046610	-20.19317	$y = 8.34 \cdot 10^{-7} \cdot x^{1.05}$
3	1.046387	-20.03556	$y = 9.30 \cdot 10^{-6} \cdot x^{1.05}$

On remarque que la régression linéaire est moins efficace pour l'algorithme Bucket : ceci explique pourquoi on trouve un ordre de x inférieur à 1 (ce qui peut à priori paraître absurde mais qui en fait traduit la faiblesse de notre régression linéaire).

Analyse théorique

a) Algorithme Merge Sort

- Meilleur cas: **O(n.log(n))**
- Pire cas: **O(n.log(n))**
- cas moyen: **O(n.log(n))**

b) Algorithme Bucket Sort

L'analyse théorique dépend du nombre de buckets mais pour un nombre de bucket buckets =n, on a

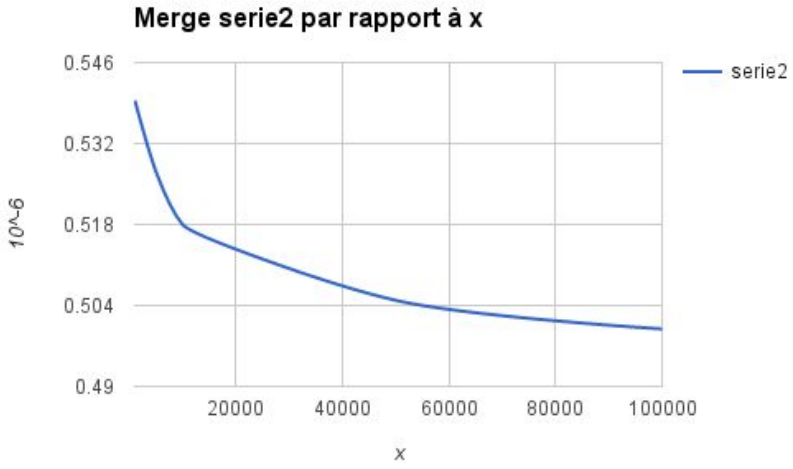
- Meilleur cas: **Ω(n)**
- Pire cas: **O(n^2)**
- cas moyen: **Θ(nln(n))**

Test du rapport

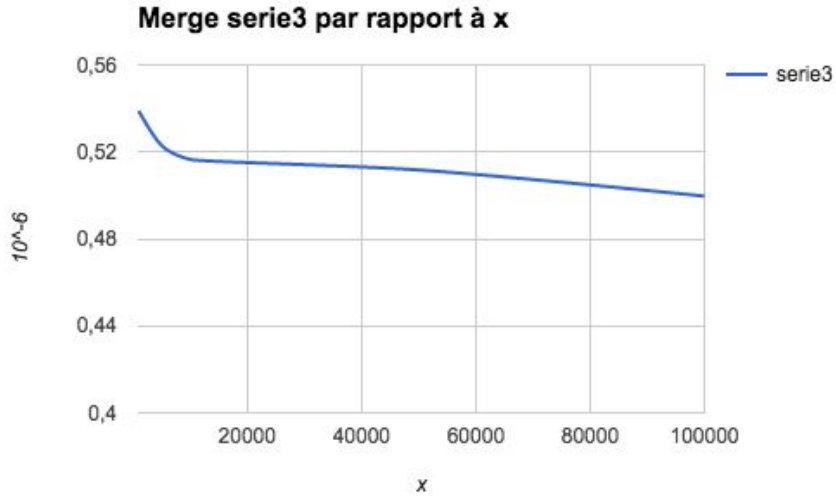
a) Algorithme Merge Sort

Pour le test du rapport dans le cas du Merge Sort, nous comparons nos valeurs a la fonction: $f(x) = x.log(x)$

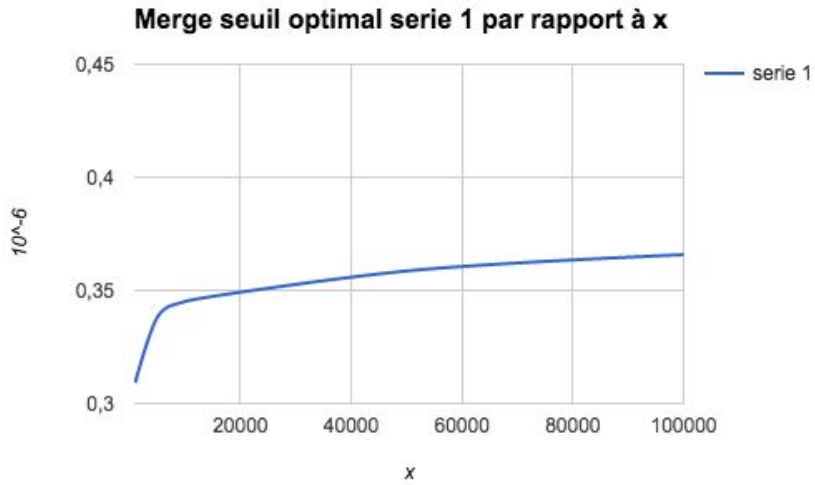
constante	0,504



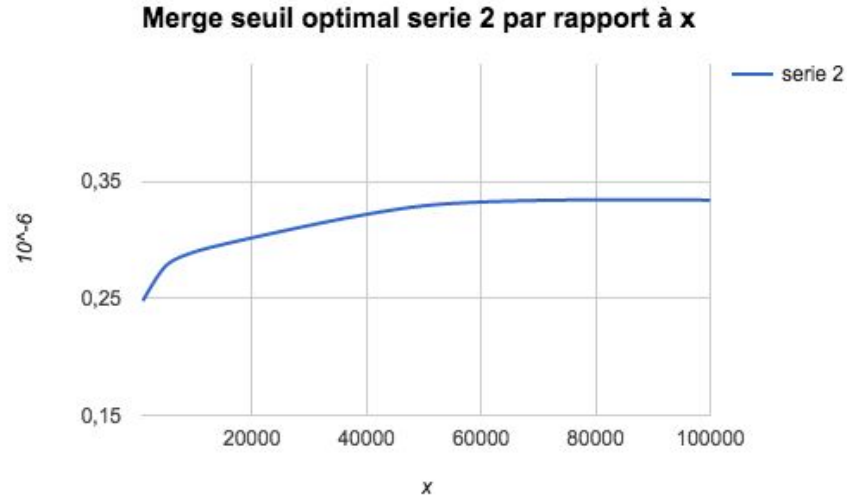
constante	0,499



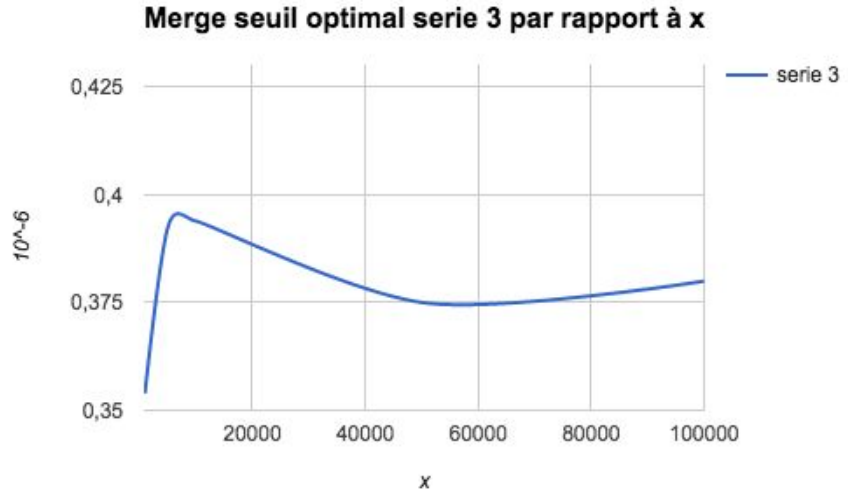
constante	0,498



constante	0,37



constante	0,33

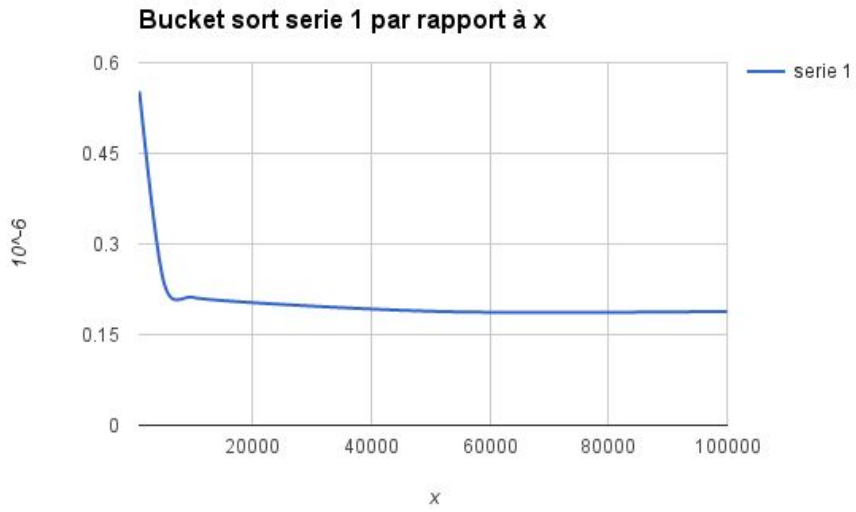


constante	0,382

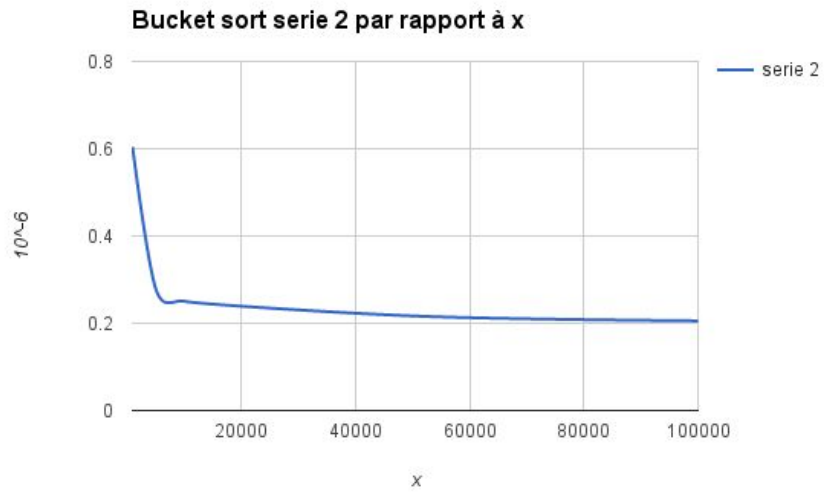
Nous constatons que les courbes convergent, cela confirme nos hypothèses théoriques et permet de faire émerger nos 2 constantes pour ces 2 seuils ≈ **0,500 & 36,1 (**. 10^{-6} **)**

b) Algorithme Bucket Sort

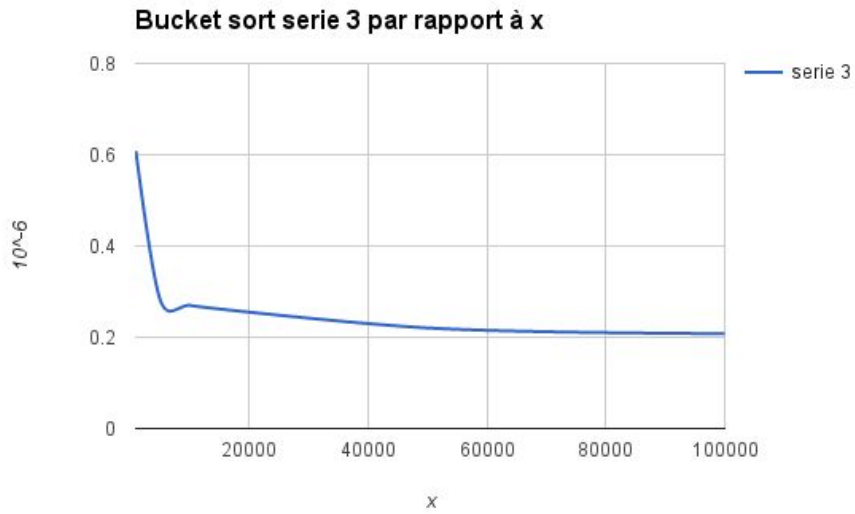
Pour le test du rapport dans le cas du Bucket Sort, nous comparons nos valeurs a la fonction $f(x) = x.log(x)$



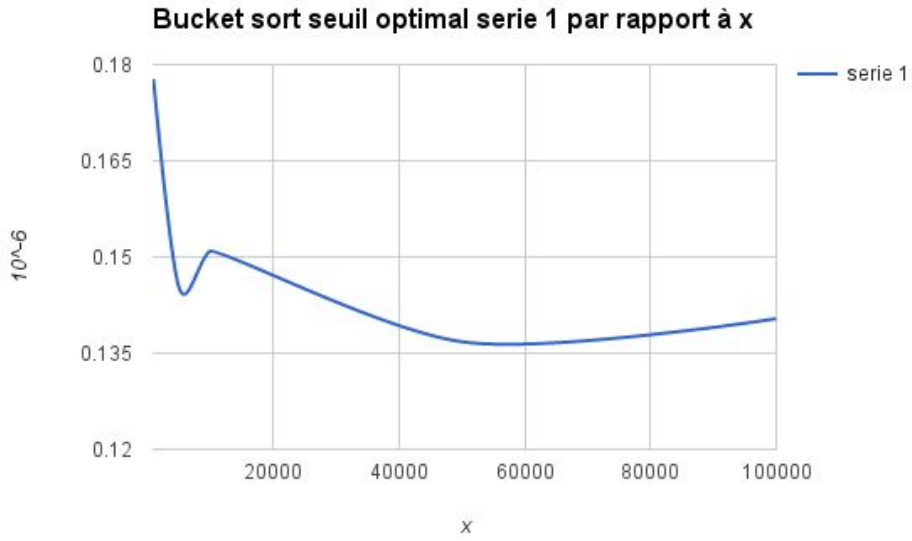
constante	0,20



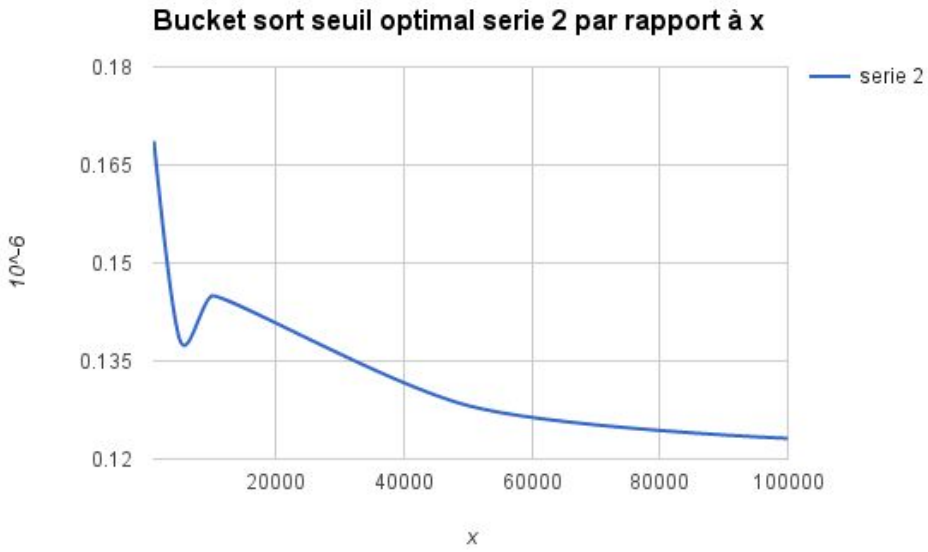
constante	0,20



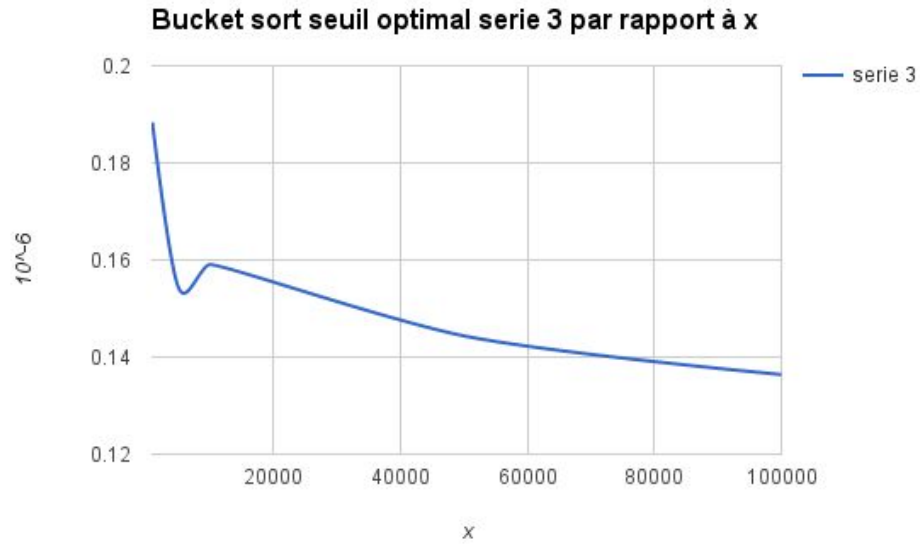
constante	0,20



constante	0,139



constante	0,122

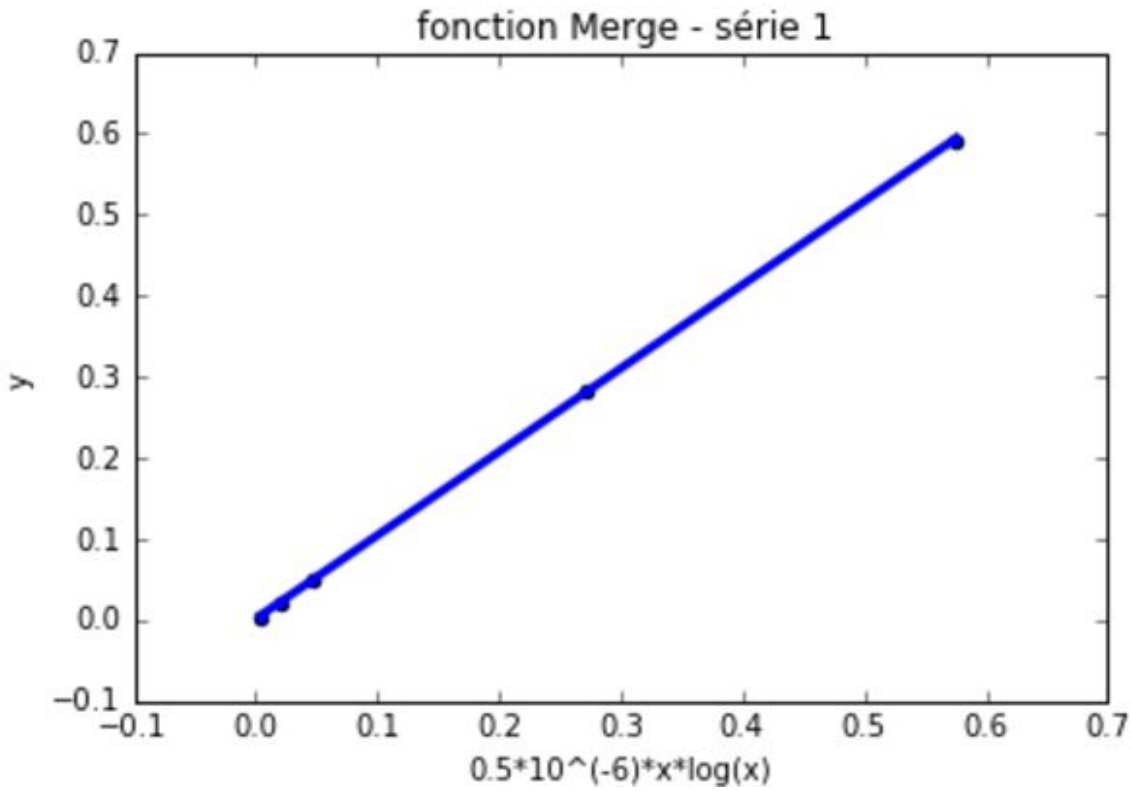


constante	0,132

Nous constatons que les courbes convergent, cela confirme nos hypothèses théoriques et permet de faire émerger nos 2 constantes pour ces 2 seuils ≈ **0,20 & 0,131 (**. 10^{-6} **)**

Test des constantes

a) Algorithme Merge Sort



Série/Merge	Pente	Coût fixe	y
1	1.030153	0.0016	$y = 5.02 \cdot 10^{-7} x ln(x) + 0.0016$
2	0.998905	0.0013	$y = 5.00 \cdot 10^{-7} x ln(x) + 0.0016$
3	0.999877	0.0017	$y = 5.00 \cdot 10^{-7} x ln(x) + 0.0017$

Série/MergeSeuil	Pente	Coût fixe	y
1	0.732339	-0.0016	$y = 3.66 \cdot 10^{-7} x ln(x) - 0.0016$
2	0.671673	-0.0025	$y = 3.36 \cdot 10^{-7} x ln(x) - 0.0025$
3	0.757983	-0.0001	$y = 3.74 \cdot 10^{-7} x ln(x) - 0.0001$

On peut remarquer que les constantes des temps d'exécution du Merge Sort sont améliorés par le MergeSeuil indépendamment du jeu de données. De plus, l'organisation du jeu de données n'influence que peu la constante de multiplication.

b) Algorithme Bucket Sort

Série/Bucket	Pente	Coût fixe	y
1	0.1860	0.0022	$y = 1.86 \cdot 10^{-7} x ln(x) + 0.0022$
2	0.1227	0.0041	$y = 1.23 \cdot 10^{-7} x ln(x) + 0.0041$
3	0.2052	0.0047	$y = 2.05 \cdot 10^{-7} x ln(x) + 0.0047$

Série/BucketSeuil	Pente	Coût fixe	y
1	0.1395	0.0002	$y = 1.40 \cdot 10^{-7} x ln(x) + 0.0002$
2	0.1227	0.0001	$y = 1.23 \cdot 10^{-7} x ln(x) + 0.0001$
3	0.1361	0.0002	$y = 1.36 \cdot 10^{-7} x ln(x) + 0.0002$

En comparaison avec le MergeSort, on voit que le BucketSeuil n'améliore pas les constantes des temps d'exécution de manière moins flagrante. En outre, il n'améliore aucunement celle de la série 2. En effet, il s'agit d'une série répartie par paquets c'est pourquoi l'utilisation d'un seuil n'influe pas sur la constante du temps d'exécution.

On remarque que les constantes du Bucket Sort sont plus intéressantes que celles du Merge Sort et ceci se traduit sur des temps d'exécution plus rapides pour le Bucket Sort. En effet, en choisissant un nombre de buckets très grand on réduit les temps d'exécution mais on alourdit la quantité de mémoire allouée à l'exécution.

Choix du seuil de récursivité

Le choix de récursivité a été discuté précédemment dans la première partie:

- Pour le Merge Sort, nous avons testé expérimentalement tous les seuil de 1 à 100 avec un seuil de 5. Le minimum est atteint avec un **seuil de 21.**
- Pour le Bucket sort nous avons considéré 3 valeurs dynamiques de nombre de buckets et le minimum global est atteint pour un **seuil de 6** avec un **nombre de bucket égale à la taille de l'exemple traité**.

Conclusion

Pour un échantillon avec une répartition uniforme des valeurs, le bucket est plus efficace car on aura une distribution d'un nombre par bucket.

De manière générale, le Bucket Sort est plus rapide mais il faut concevoir qu'il est plus gourmand en mémoire usager que le Merge Sort à cause de la création de n buckets à chaque étape. L'exécution de Bucket sort étant effectué sur des postes avec beaucoup de ressources, cette consommation n'est pas un problème dans notre cas.

Cependant, pour des échantillons avec une répartition plus chaotique, l'algorithme Merge Sort est à favoriser, en effet sa complexité est bornée et nous assure donc un résultat moyen et en pire cas acceptable.

Source

Algorithmes:

- http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#Python
- http://stackoverflow.com/questions/25690175/bucket-sort-faster-than-quicksort

Régression linéaire :

bibliothèque sklearn de Python