Rapport TP1 INF4705

Florentin Burgeat (1809123), Elliot Sisteron (1807165) 23 octobre 2015

Introduction

Ce rapport a pour but de présenter les résultats de nos analyses lors de nos implantations des algorithmes de tri à bulle et de tri rapide programmés en langage C++.

En effet, nous allons adopter dans ce rapport une approche dite "hybride" afin de nous orienter vers une fonction précise évaluant le temps d'exécution en fonction de la taille de l'exemplaire.

Une analyse théorique ne nous permettant de ne trouver que l'ordre de grandeur de la complexité de l'algorithme, l'approche hybride permettra de déterminer les constantes en jeu dans nos implantations pour préciser cette grandeur.

On effectuera donc d'abord un rappel des complexité théoriques des algorithmes étudiés. Puis, nous évaluerons le seuil de récursivité optimal pour l'algorithme de tri rapide. Nous développerons ensuite l'analyse en effectuant les trois tests (de puissance, du rapport, et des constantes) sur nos résultats afin de trouver les équations correspondante. Finalement, nous déterminerons quel algorithme est le plus efficace pour résoudre des problèmes de tri.

Table des matières

1 Analyse théorique des deux algorithmes étudiés

On rappelle ici le fonctionnement des deux tris étudiés ainsi qu'une analyse théorique de leur complexité.

1.1 Tri à bulle

Le tri à bulle est probablement un des tris les plus simple à comprendre. L'algorithme, si on l'énonce en haut niveau, consiste à parcourir le tableau en échangeant deux éléments adjacents dès qu'il y a une incohérence ; et cela jusqu'à ce que le tableau soit trié.

1.1.1 La variante du tri shaker

Cette variante du tri à bulle consiste à parcourir le tableau plus intelligemment. En effet, le tri à bulle tient son nom du constat suivant : au fur et à mesure des itérations, les plus grand éléments sont « remontés » vers leur position adéquate (comme des bulles...). Le fait est que, à chaque itération, le maximum courant se retrouve à la bonne place ; il est donc inutile d'effectuer des comparaisons sur les éléments déjà correctement placés. Il s'agit là d'une première optimisation réduisant la durée du parcours.

De même, si l'on effectue le trajet dans l'autre sens, les plus petits éléments sont « descendus ». On voit donc ici une seconde optimisation : en parcourant le tableau dans un sens, puis dans l'autre, on tri le tableau plus rapidement. Encore une fois, on s'agenouille devant la débordante imagination des informaticiens : le nom du tri renvoie directement à ce mouvement de va-et-vient (non, ils n'avaient certainement pas pensé à un autre nom, aheum...).

1.1.2 Complexité

Pour ce qui est de l'algorithme du tri à bulle et sa variante shaker, leur complexité temporelle $T(n)$ est :

- en meilleur cas : $T \in \Theta(n)$, lorsque le tableau est déjà trié, il suffit de le parcourir une fois pour s'en rendre compte ;
- en moyen cas : $T \in \Theta(n^2)$, en supposant, à chaque tour de boucle, que le maximum des valeurs non-remontées ait une position tiré au sort de manière uniforme dans le tableau ;
- en pire cas : $T \in \Theta(n^2)$, dans le cas où le tableau est inversement trié, il faut faire remonter les éléments un à un (ce qui a un coup linéaire à chaque fois).

1.2 Tri rapide

Le tri rapide, plus connu sous le nom de QuickSort ou encore tri de Hoare, est un des tris les plus plus efficace (en prenant en compte quelques améliorations que nous allons présenter). Il est fondé sur principe « Diviser pour régner ». L' « intelligence » de ce tri se situe au niveau de la décomposition de l'exemplaire en sous-exemplaires avant l'appel récursif. En choisissant un élément pivot dans le tableau, on partitionne ce dernier en plaçant les éléments inférieurs à gauche, et les autres, supérieurs, à droite. En effectuant cette démarche récursivement sur chaque sous tableau, on fini par le trier.

1.2.1 Choix du pivot

Une première variante du tri rapide se situe dans le choix du pivot : selon la position du pivot après la partition, on peut avoir deux sous tableaux de taille plus ou moins équilibrées. Plus ces deux sous-tableaux ont une taille similaire, plus on aura « décomposé » le problème. De fait, l'efficacité de cet algorithme repose grandement sur le choix du pivot.

Nous étudierons principalement le choix de pivot simple (premier élément du tableau à partitionner) ou aléatoire (tiré au sort de manière uniforme parmis les éléments du tableau).

1.2.2 Seuil de recursivité

Une deuxième variante consiste à utiliser un algorithme de tri simple lorsque l'on arrive à un appel du QuickSort sur un tableau assez petit. Cela peut améliorer grandement les performances, pour deux raisons :

— un tri simple peut être plus efficace sur des exemplaires de petites tailles compte-tenu des constantes multiplicatives cachées ;
— cela limite les ajouts sur la pile d'appels récursifs, ce qui peut parfois causer des ralentissements lorsque l'on travaille sur de gros exemplaires.

La difficulté est alors de trouver le seuil de récursivité qui améliore globalement les performances de notre algorithme. Cela dépend beaucoup de l'implantation et nécessite donc une approche empririque. Nous travaillerons avec le tri simple du shaker, mais on peut noter qu'une variante du tri rapide, appelé « l'Introspective Sort » utilise le tri par tas.

1.2.3 Complexité

La complexité temporelle $T(n)$ du QuickSort est :

— en meilleur cas : $T \in \Theta(n \log(n))$;
— en moyen cas : $T \in \Theta(n \log(n))$, comme on l'a démontré dans le cours ;
— en pire cas : $T \in \Theta(n^2)$, dans le cas où le tableau est déjà trié, le choix du pivot simple entraîne cette complexité ; avec un choix de pivot aléatoire, on espère empêcher ce cas de survenir.

On s'attend, avec nos améliorations, à réduire de manière notable la constante multiplication cachée dans ces notations.

2 Détermination empirique de la complexité

2.1 Quelques remarques avant de commencer

2.1.1 Cadre expérimental

Nous avons principalement travaillé sur un MacBook Air mi-2011, dont la configuration est la suivante :

FIGURE 1 – Cadre expérimental

Le langage choisi est le C++ et compilateur utilisé est g++.

2.1.2 Jeux de données

Nous utiliserons les jeux de données fournit dans le cadriciel du TP. Nous ne travaillerons pas avec les exemplaires de taille 500000, car une seule exécution du BubbleSort sur un de ces fichier prend environ 25 minutes.

2.1.3 Notations

On considérera la notation suivante pour la suite du TP : pour un exemplaire i, le couple (x_i, y_i) désignera la taille de l'exemplaire x_i étudié et le temps d'exécution de l'algorithme sur l'exemplaire y_i.

2.1.4 Seuil de recursivité

Nous avons choisi comme seuil de récursivité 8. Nous avons considéré les QuickSort aléatoire et simple avec des seuils variant de 0 à 80. Puis, nous les avons appliqué chacun sur l'ensemble des exemplaires 10 fois (pour éviter d'avoir trop de bruit du à l'aléatoire).

Il nous a suffit de regarder quels tris prenaient le moins de temps pour effectuer cela, et il s'avère que la constante 8 est un bon compromis sur notre machine.

2.2 Présentation du programme réalisé

Le programme C++ réalisé se trouve dans le dossier du rapport.

2.2.1 Installation et usage

L'ensemble des instructions à suivre pour installer, exécuter et utiliser notre programme sur une distribution Linux ou MacOS se trouve dans le fichier *README.txt*.

2.2.2 Explications

Nous avons programmé de manière générique [1] les algorithmes BubbleSort et QuickSort. Puis, en passant en paramètre des adresses de fonctions et des valeurs (fonction de choix de pivot, valeur du seuil de recursivité...), nous avons pu être en mesure d'influencer certaines parties de leur fonctionnement intrinsèque.

Par exemple, pour le QuickSort,

Listing 1 – Notre implantation du QuickSort générique

```
1  // Generic and partial quick sort
2  void quick_sort_partial(int* t, int start, int end, int recursive_limit, pivot_generator ↩
       get_pivot) {
3      if(end-start > recursive_limit) {
4          int pivot_position = partition(t, start, end, get_pivot);
5          quick_sort_partial(t, start, pivot_position-1, recursive_limit, get_pivot);
6          quick_sort_partial(t, pivot_position+1, end, recursive_limit, get_pivot);
7      }
8      else if (start < end) {
9          bubble_sort_partial(t, start, end, shaker_bounds);
10     }
11 }
12
13 // Generic, full, quick sort
14 void quick_sort_generic(int* t, int n, int recursive_limit, pivot_generator get_pivot){
15     quick_sort_partial(t, 0, n-1, recursive_limit, get_pivot);
16 }
17
18 // The random pivot function
19 int random_pivot(int start, int end) {
20     return (start + int((end-start+1)*rand()/(RAND_MAX + 1.0)));
21 }
22
23 // An example of a call for the random pivot optimization
24 void quick_sort_random_pivot(int* t, int n) {
25     quick_sort_generic(t, n, 0, random_pivot);
26 }
```

Nous avons ensuite définit un type de fonction pour les tris :

Listing 2 – Typedef pour les tris

```
1  // Typedef of a sorting function
2  typedef void(*sort_function)(int*, int);
```

Puis, nous avons mis les tris que nous souhaitions étudier dans un tableau :

Listing 3 – Typedef pour les tris

```
1  const sort_function sort_functions[NUMBER_OF_SORTS] = {bubble_sort, shaker_sort, quick_sort,↩
       quick_sort_random_pivot, quick_sort_recursive_stop, ↩
       quick_sort_random_pivot_recursive_stop, std_sort};
```

1. On entend par là que le fonctionnement de l'algorithme est facilement modulable, non pas qu'il fonctionne avec des types génériques

En itérant dans ce tableau sur les fichiers d'input, nous avons donc pu tester tous le tris d'une traite.

2.2.3 Résultats du programme

En sortie de notre programme et après traitement excel, nous avons les valeurs suivantes :

0->9	1000	5000	10000	50000	100000
BubbleSort	6,09E-03	1,60E-01	6,06E-01	1,60E+01	6,15E+01
ShakerSort	3,79E-03	9,73E-02	3,62E-01	9,53E+00	3,65E+01
QuickSort	1,17E-04	6,54E-04	1,49E-03	9,33E-03	1,71E-02
QSRandomPivot	1,21E-04	6,80E-04	1,82E-03	9,46E-03	1,77E-02
QSRecursiveStop	1,47E-04	6,54E-04	1,59E-03	8,45E-03	1,77E-02
QSRandomPivotRecursiveStop	1,18E-04	6,54E-04	1,56E-03	8,28E-03	1,69E-02
STD Sort	5,65E-05	2,56E-04	6,04E-04	3,10E-03	6,51E-03
10->19	**1000**	**5000**	**10000**	**50000**	**100000**
BubbleSort	3,09E-03	8,09E-02	3,09E-01	7,97E+00	3,33E+01
ShakerSort	6,76E-04	1,93E-02	7,15E-02	1,75E+00	6,80E+00
QuickSort	1,37E-04	8,23E-04	1,78E-03	1,01E-02	2,08E-02
QSRandomPivot	1,44E-04	8,28E-04	1,71E-03	1,13E-02	2,03E-02
QSRecursiveStop	1,46E-04	7,85E-04	1,57E-03	1,00E-02	1,94E-02
QSRandomPivotRecursiveStop	1,39E-04	7,88E-04	1,71E-03	1,05E-02	2,10E-02
STD Sort	2,72E-05	1,38E-04	2,79E-04	1,76E-03	3,38E-03
20->29	**1000**	**5000**	**10000**	**50000**	**100000**
BubbleSort	7,88E-03	1,59E-01	6,25E-01	1,55E+01	6,19E+01
ShakerSort	5,14E-03	1,18E-01	4,49E-01	1,18E+01	4,42E+01
QuickSort	1,22E-04	7,56E-04	1,57E-03	9,63E-03	2,09E-02
QSRandomPivot	1,30E-04	8,35E-04	1,70E-03	1,03E-02	2,17E-02
QSRecursiveStop	1,27E-04	7,74E-04	1,71E-03	9,77E-03	2,07E-02
QSRandomPivotRecursiveStop	1,27E-04	8,12E-04	1,71E-03	1,02E-02	2,09E-02
STD Sort	2,45E-05	1,32E-04	2,72E-04	1,63E-03	3,27E-03

FIGURE 2 – Résultat du programme

2.3 Test de puissance

Soit un ensemble d'exemplaires appliqués à une implantation d'un algorithme. Le test de puissance consiste à essayer de faire passer une droite approchant les points $(x_i, y_i)_i$ sur un graphique à l'échelle log-log.

Une droite des moindres carrées se prête bien à cet effet, et le coefficient R^2 nous permet d'évaluer la qualité de cette évaluation. Cette droite peut s'exprimer de la manière suivante $log_a(y) = blog_a(x) + m$. Si la droite est cohérente, alors on peut exprimer T par $T(n) = An^B$, avec $A = a^b$ et $B = m$.

Pour effectuer le test de puissance, nous avons passé au log_{10} nos coordonnées $(x_i, y_i)_i$. Puis, nous avons effectuer une régression linéaire à l'aide du progarmme GeoGebra. On a les résultats suivants :

— Pour les exemplaires de 0 à 9 :

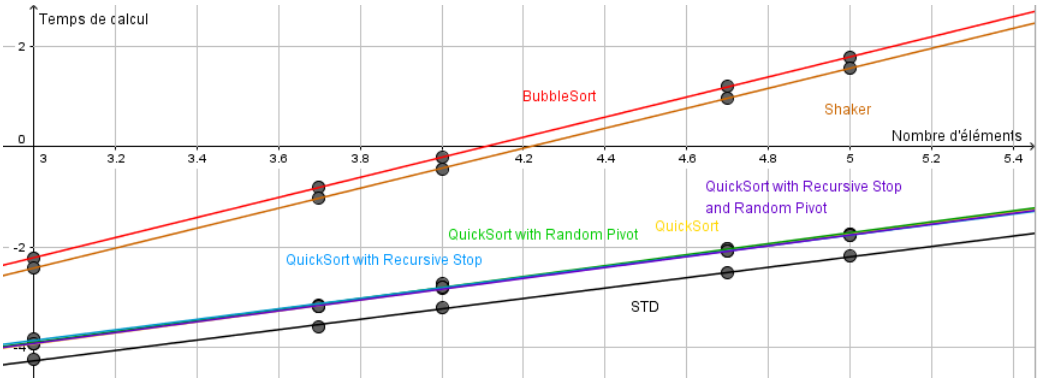

FIGURE 3 – Test de puissance sur les exemplaires 0 à 9

$$\text{BubbleSort}(x) = 0.000000006009954\, x^{2.003626303676326}$$
$$\text{QSRandPRecStop}(x) = 0.000000067998541\, x^{1.081678022975722}$$
$$\text{QSRandomPivot}(x) = 0.000000065728015\, x^{1.093761304752733}$$
$$\text{QSRecursiveStop}(x) = 0.000000096986387\, x^{1.050379022163327}$$
$$\text{QuickSort}_1(x) = 0.000000059057112\, x^{1.098320054986194}$$
$$\text{STD}_1(x) = 0.000000041359388\, x^{1.037526215721666}$$
$$\text{ShakerSort}(x) = 0.000000003983672\, x^{1.993604685621617}$$

FIGURE 4 – Constantes trouvées par le test de puissance sur les exemplaires 0 à 9

— Pour les exemplaires de 10 à 19 :

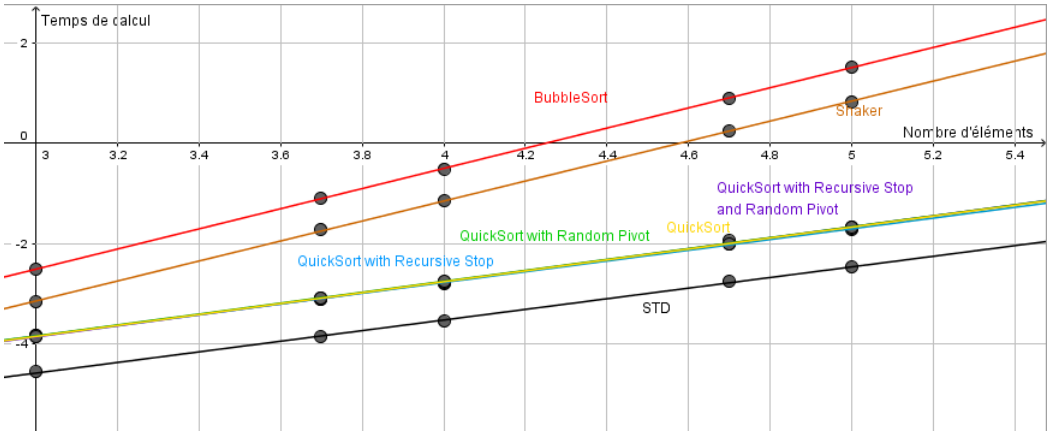

FIGURE 5 – Test de puissance sur les exemplaires 10 à 19

$BubbleSort2(x) = 0.000000002844896\, x^{2.012038276021111}$
$QSRandPRecStop2(x) = 0.000000070214287\, x^{1.09735680477542}$
$QSRandomPivot2(x) = 0.000000077079943\, x^{1.09007362957686}$
$QSrecursiveStop2(x) = 0.000000086044061\, x^{1.072133725045826}$
$QuickSort2(x) = 0.000000074803312\, x^{1.091057282082321}$
$STD2(x) = 0.000000017157364\, x^{1.060020215547512}$
$ShakerSort2(x) = 0.000000000740196\, x^{1.995232037704739}$

FIGURE 6 – Constantes trouvées par le test de puissance sur les exemplaires 10 à 19

— Pour les exemplaires de 20 à 29 :

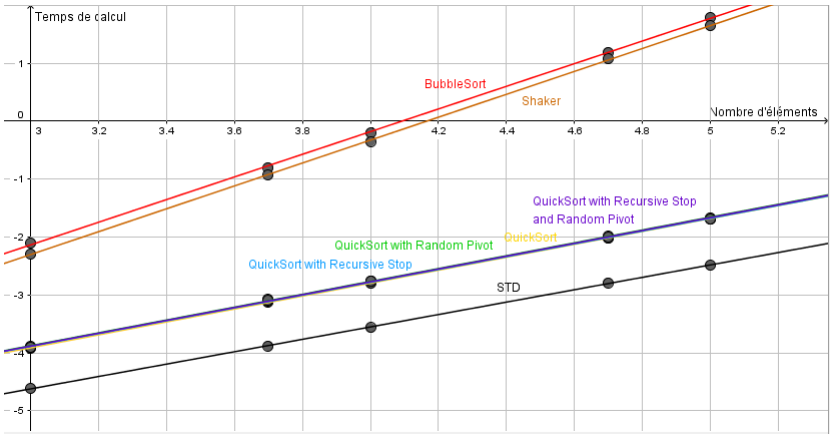

FIGURE 7 – Test de puissance sur les exemplaires 20 à 29

$BubbleSort3(x) = 0.000000010098232\, x^{1.953854113767465}$
$QSRandPRecStop3(x) = 0.000000006222299\, x^{1.108217199017251}$
$QSRandomPivot3(x) = 0.000000062857453\, x^{1.109111596032974}$
$QSRecursiveStop3(x) = 0.000000062695036\, x^{1.105270430466943}$
$QuickSort3(x) = 0.000000055310832\, x^{1.115333356782214}$
$STD3(x) = 0.000000014870595\, x^{1.069196328166457}$
$ShakerSort3(x) = 0.000000005950848\, x^{1.974684761416813}$

FIGURE 8 – Constantes trouvées par le test de puissance sur les exemplaires 20 à 29

2.3.1 Discussion des résultats

Pour les trois cas, on arrive à faire passer une droite des moindres carrés avec un coefficient de régression R^2 d'au moins 99.8% (oui monsieur !).

Mais ? Pourtant la consommation du tri rapide ne s'exprime pas sous la forme $T(n) = An^B \ldots$ Cela vient du fait que le log tend vers une constante sur graphique à l'échelle log-log et influe peut la courbe. De ce fait, on obtient une consommation qui s'apparente à $T(n) = An^B$ avec $B \approx 1$.

On peut alors passer au test du rapport vu que l'on a une meilleure idée la forme $f(n)$ des consommations à spécifier. Cela va nous permettre de valider cette hypothèse et préciser les constantes mises en jeu.

2.4 Test du rapport

Le test du rapport consiste à essayer tracer les points $(x_i, y_i/f(x_i))_i$, avec f l'ordre exact supposé de notre consommation. Si l'on tend vers une constante, cette hypothèse est confirmé et l'on peut spécifier la constante multiplicative.

2.4.1 Tri à Bulle

En faisant le test du rapport avec les complexités théoriques du tri à bulle on obtient les courbes suivantes :

— En meilleur cas :

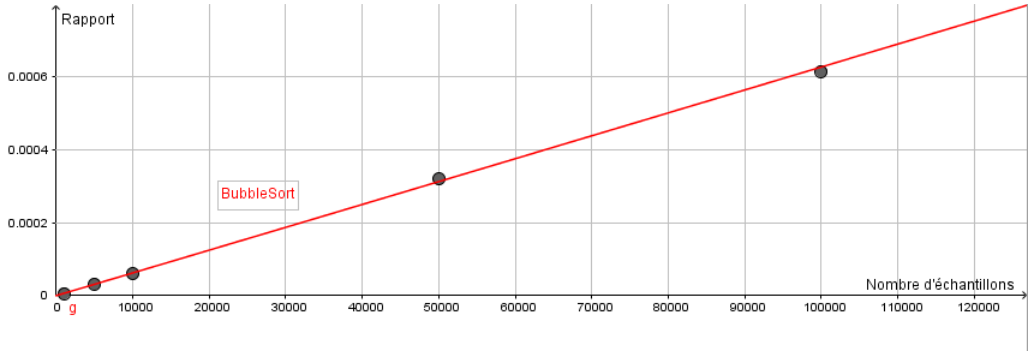

FIGURE 9 – Test du rapport en meilleur cas, bubble sort, exemplaires 0 à 9

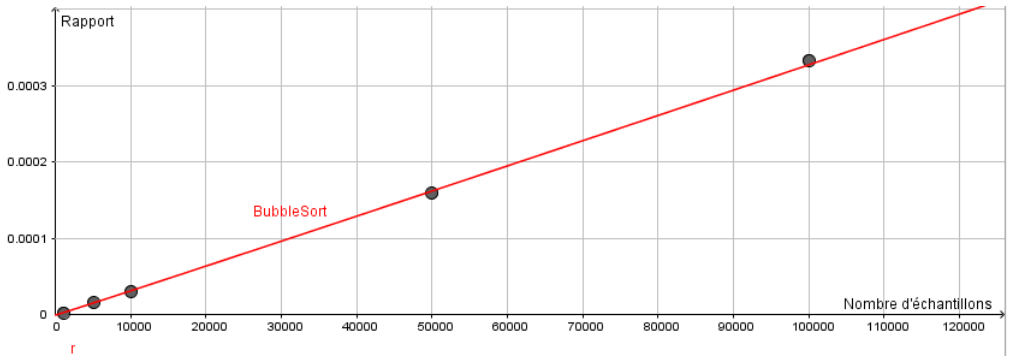

FIGURE 10 – Test du rapport en meilleur cas, bubble sort, exemplaires 10 à 19

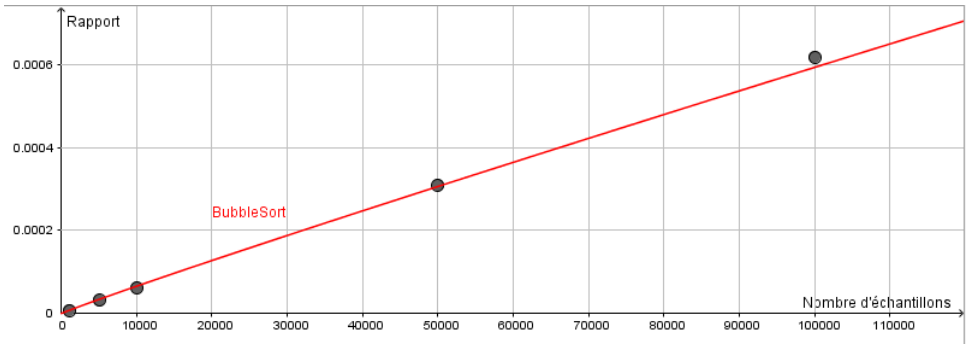

FIGURE 11 – Test du rapport en meilleur cas, bubble sort, exemplaires 20 à 29

— En moyen et pire cas :

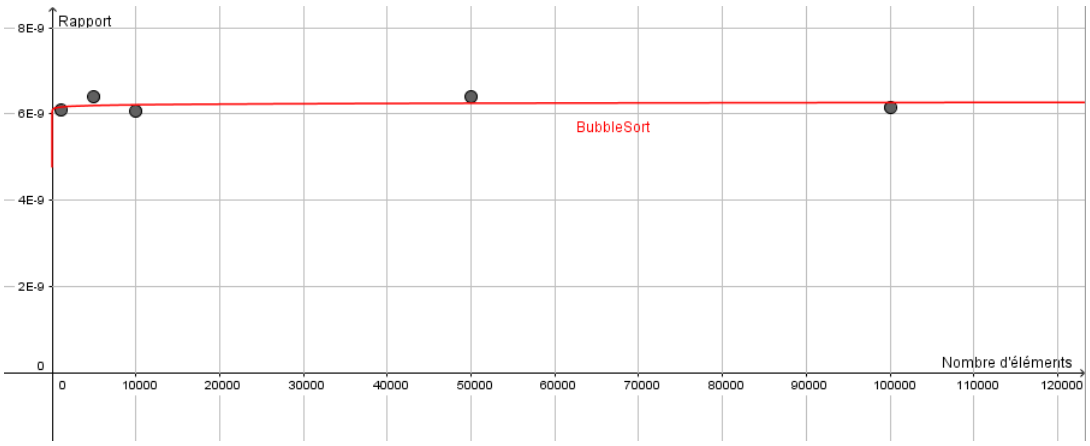

FIGURE 12 – Test du rapport en moyen/pire cas, bubble sort, exemplaires 0 à 9

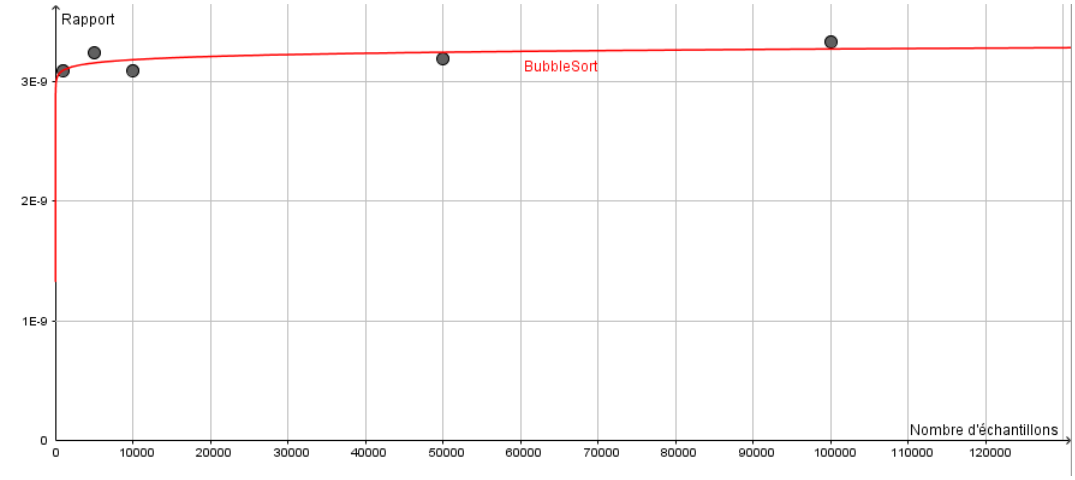

FIGURE 13 – Test du rapport en moyen/pire cas, bubble sort, exemplaires 10 à 19

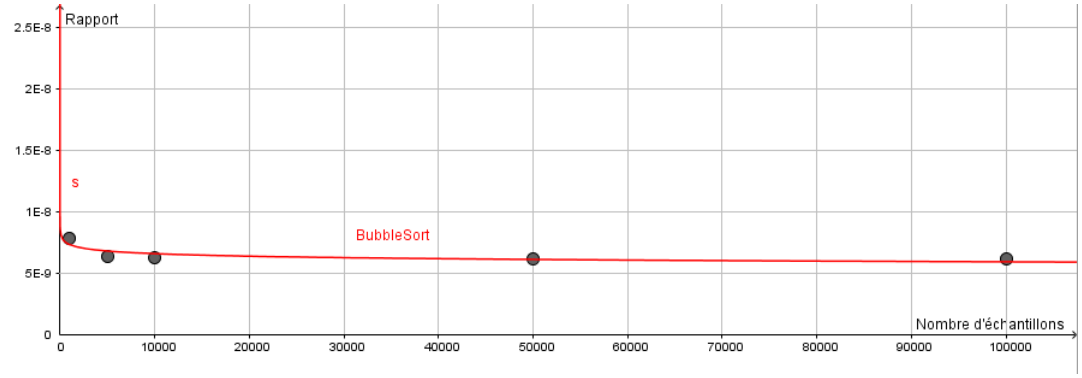

FIGURE 14 – Test du rapport en moyen/pire cas, bubble sort, exemplaires 20 à 29

Pour chaque série d'exemplaire, on remarque que la courbe d'analyse en moyen et pire cas converge vers une constante quand on choisi l'analyse théorique en n^2. La constante multiplicative se trouve être entre $3 \cdot 10^{-9}$ et $6 \cdot 10^{-9}$.

2.4.2 Tri rapide

Le tri rapide s'effectue théoriquement en $\mathcal{O}(n \log n)$ en meilleur et moyen cas, et en $\mathcal{O}(n^2)$ en pire cas.

Nous avons donc effectué le test du rapport sur chacun des quatre versions du Tri rapide et chacune des séries. On trouve les résultats suivant :

1. Tri Rapide Simple
   (a) En meilleur et moyen cas :

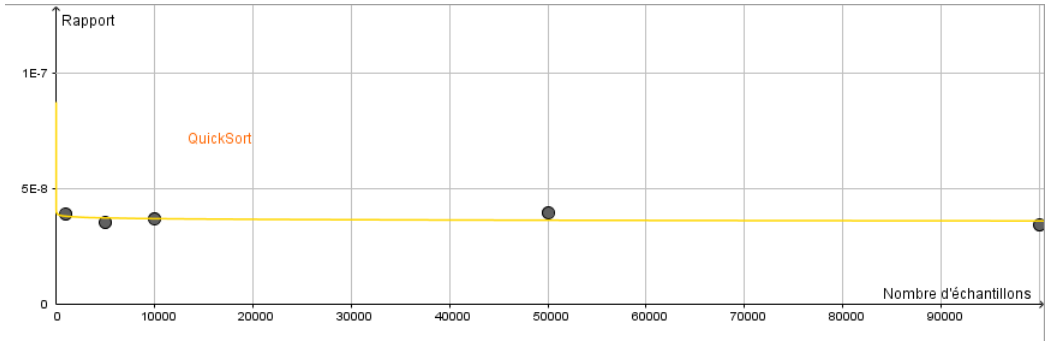

FIGURE 15 – Test du rapport en meilleur/moyen cas, quick sort simple, exemplaires 0 à 9

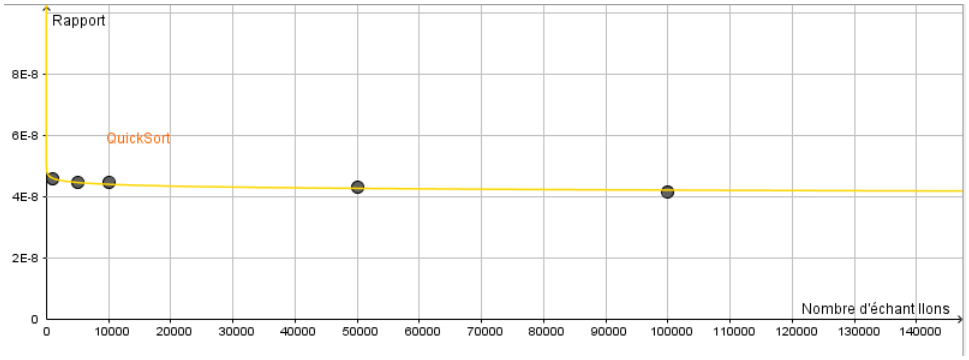

FIGURE 16 – Test du rapport en meilleur/moyen cas, quick sort simple, exemplaires 10 à 19

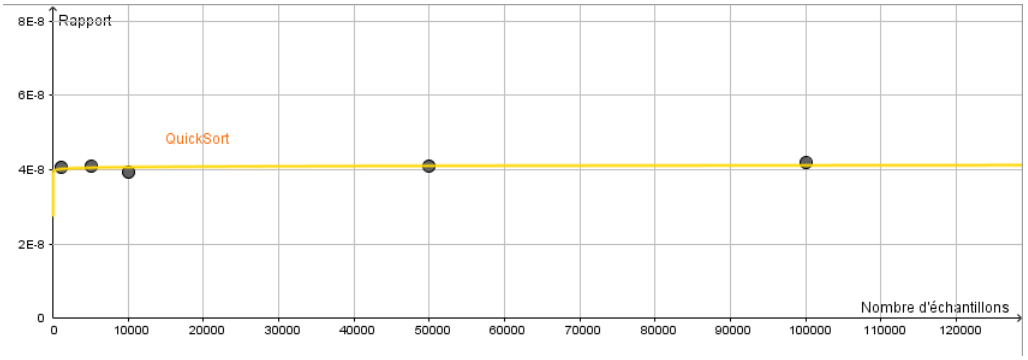

FIGURE 17 – Test du rapport en meilleur/moyen cas, quick sort simple, exemplaires 20 à 29

(b) En pire cas :

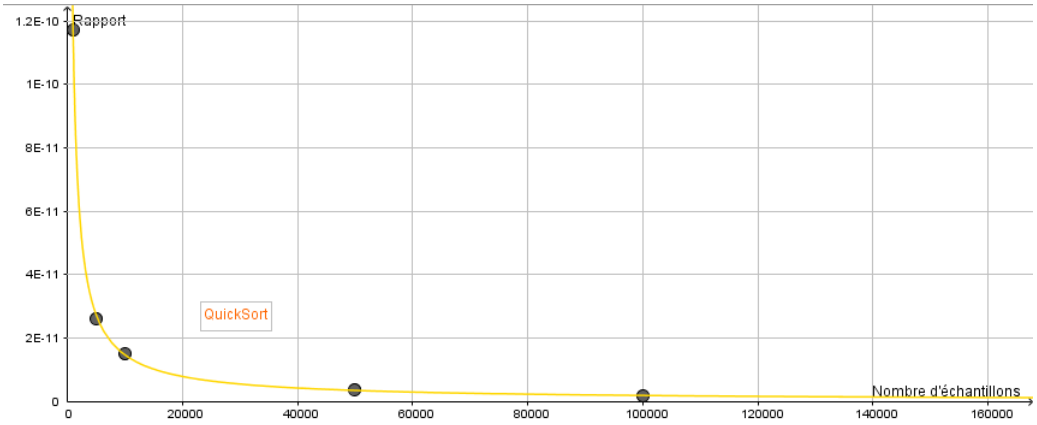

FIGURE 18 – Test du rapport en pire cas, quick sort simple, exemplaires 0 à 9

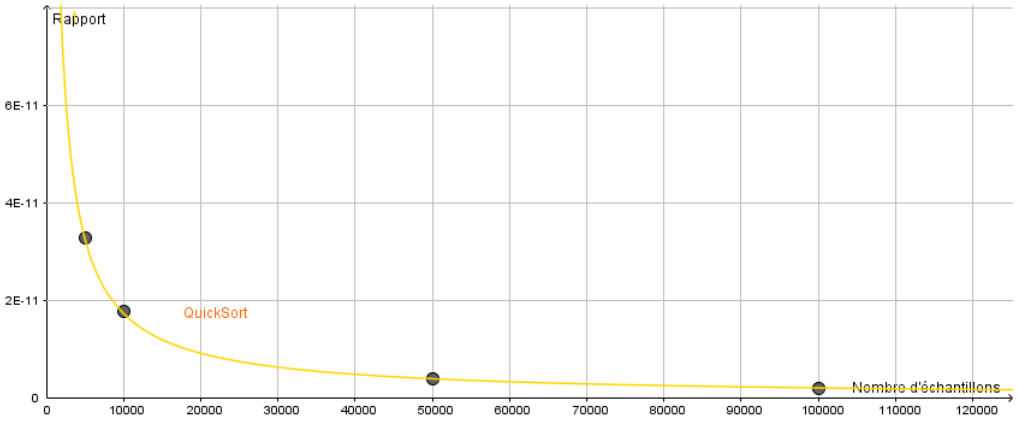

FIGURE 19 – Test du rapport en pire cas, quick sort simple, exemplaires 10 à 19

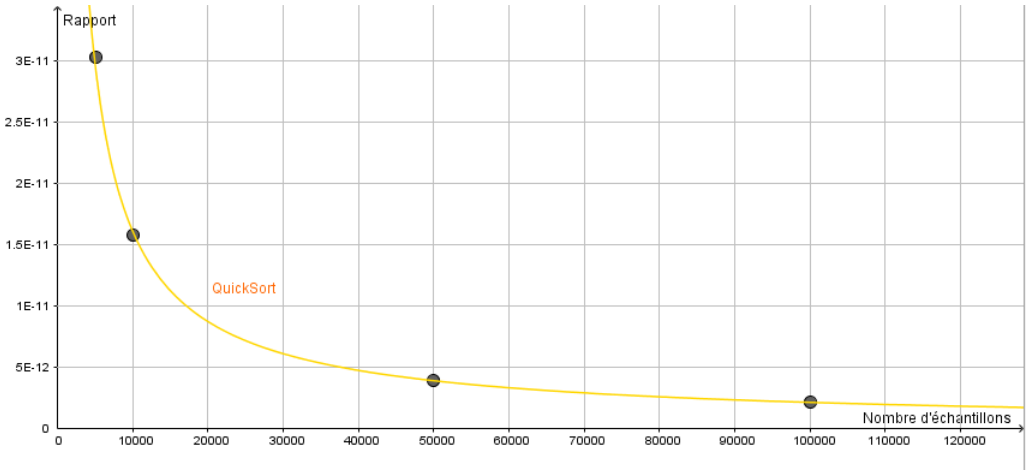

FIGURE 20 – Test du rapport en pire cas, quick sort simple, exemplaires 20 à 29

2. Tri rapide avec un pivot aléatoire

   (a) En meilleur et moyen cas :

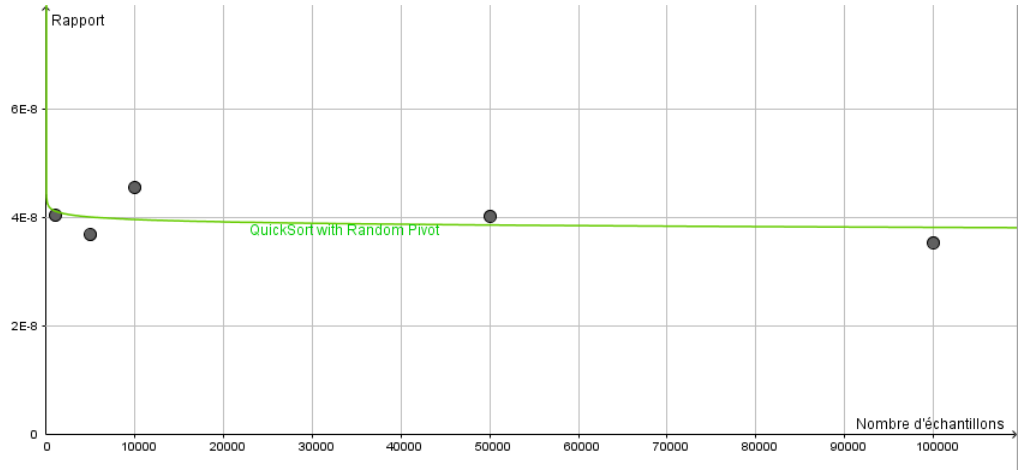

FIGURE 21 – Test du rapport en meilleur/moyen cas, quick sort à pivot aléatoire, exemplaires 0 à 9

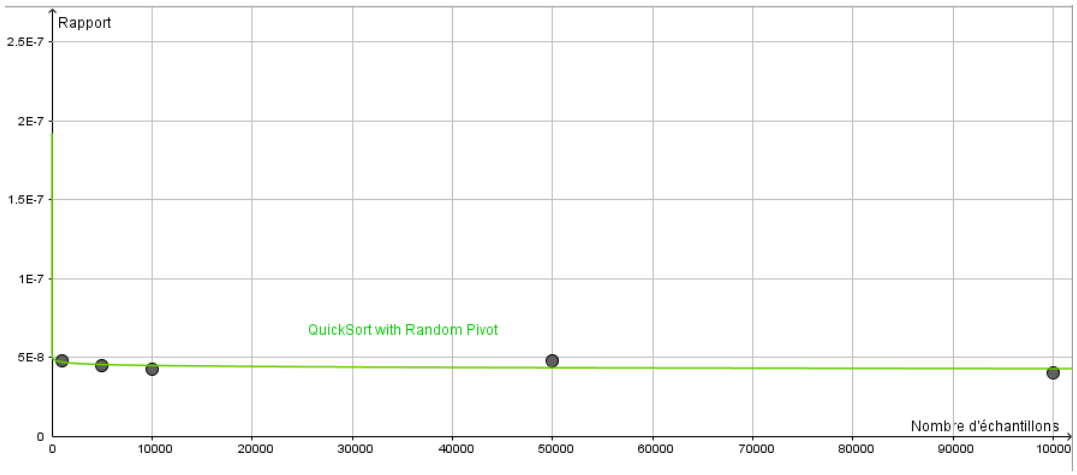

FIGURE 22 – Test du rapport en meilleur/moyen cas, quick sort à pivot aléatoire, exemplaires 10 à 19

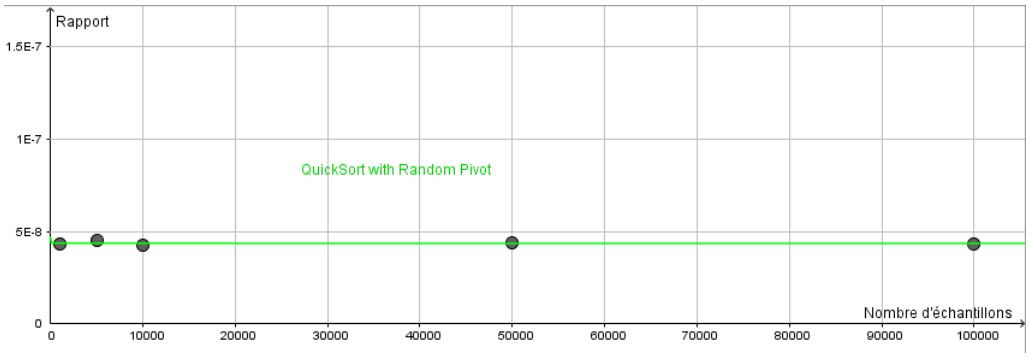

FIGURE 23 – Test du rapport en meilleur/moyen cas, quick sort à pivot aléatoire, exemplaires 20 à 29

(b) En pire cas :

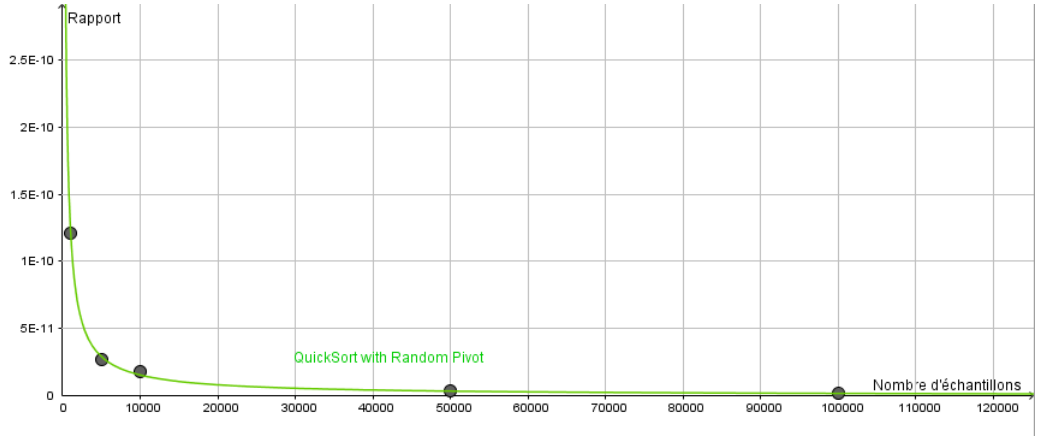

FIGURE 24 – Test du rapport en pire cas, quick sort à pivot aléatoire, exemplaires 0 à 9

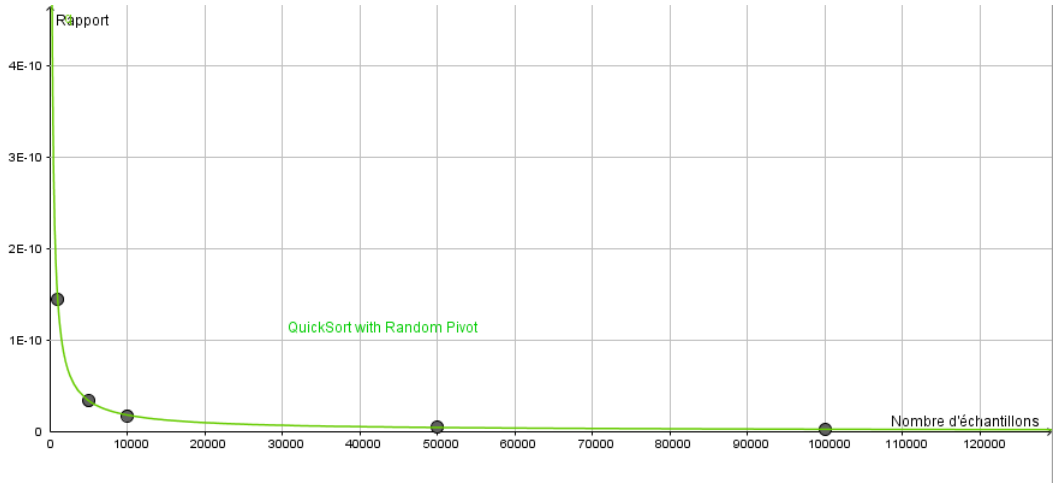

FIGURE 25 – Test du rapport en pire cas, quick sort à pivot aléatoire, exemplaires 10 à 19

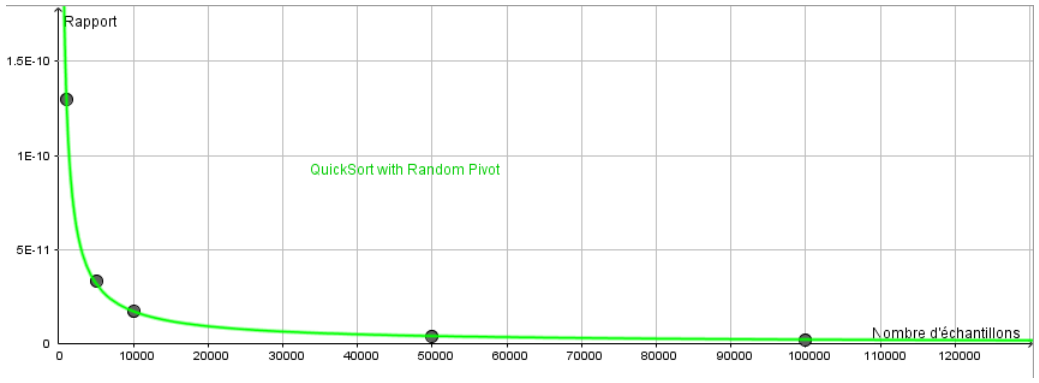

FIGURE 26 – Test du rapport en pire cas, quick sort à pivot aléatoire, exemplaires 20 à 29

3. Tri rapide avec un seuil de récursivité

   (a) En meilleur et moyen cas :

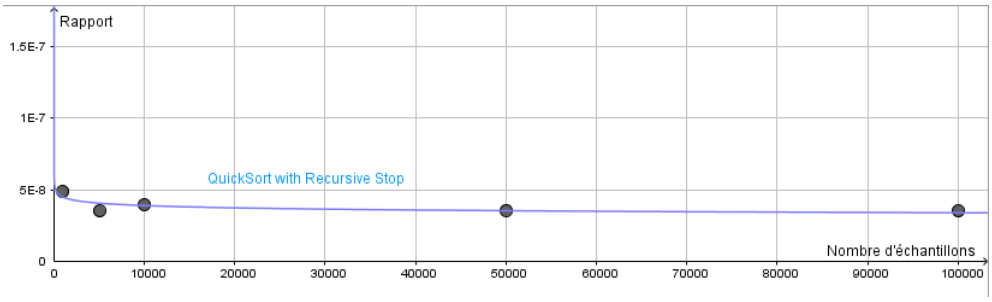

FIGURE 27 – Test du rapport en meilleur/moyen cas, quick sort à seuil, exemplaires 0 à 9

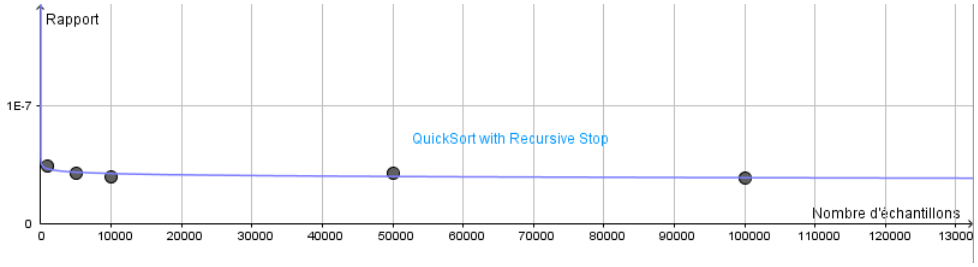

FIGURE 28 – Test du rapport en meilleur/moyen cas, quick sort à seuil, exemplaires 10 à 19

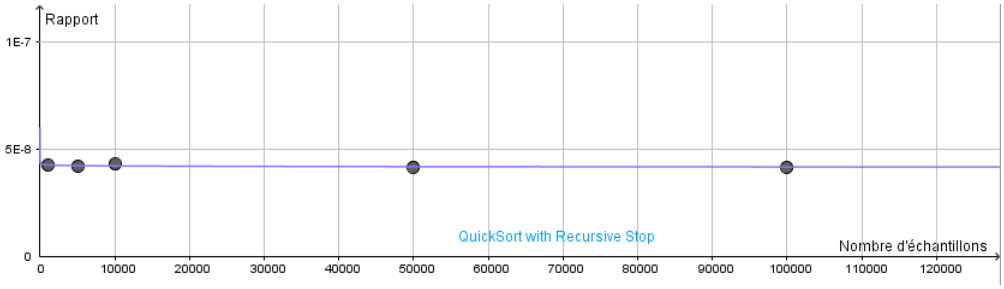

FIGURE 29 – Test du rapport en meilleur/moyen cas, quick sort à seuil, exemplaires 20 à 29

(b) En pire cas :

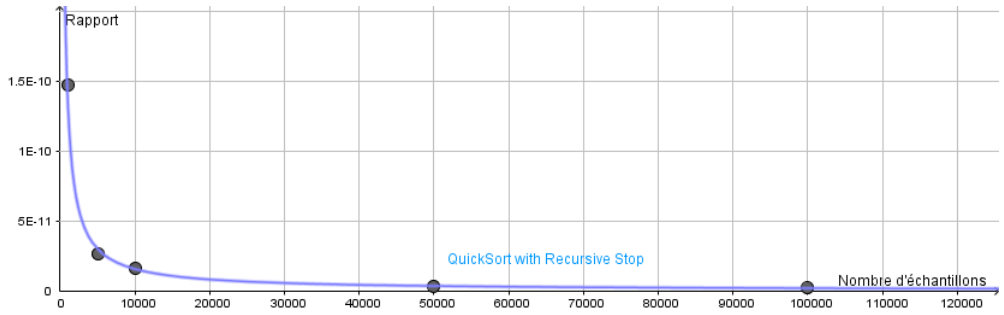

FIGURE 30 – Test du rapport en pire cas, quick sort à seuil, exemplaires 0 à 9

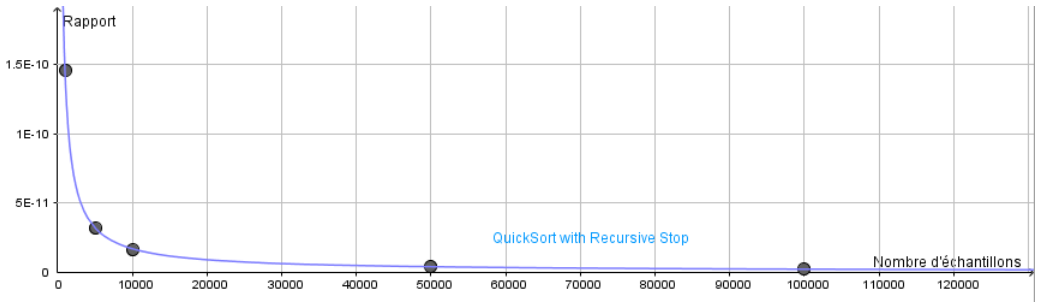

FIGURE 31 – Test du rapport en pire cas, quick sort à seuil, exemplaires 10 à 19

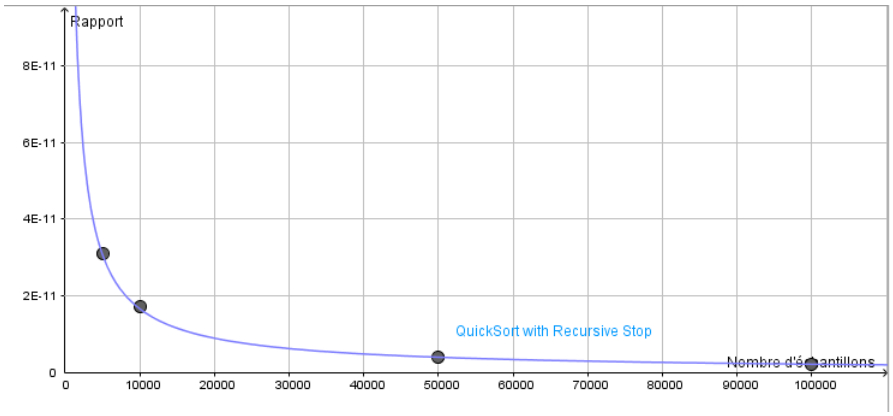

FIGURE 32 – Test du rapport en pire cas, quick sort à seuil, exemplaires 20 à 29

4. Tri rapide avec un pivot aléatoire et un seuil de récursivité

   (a) En meilleur et moyen cas :

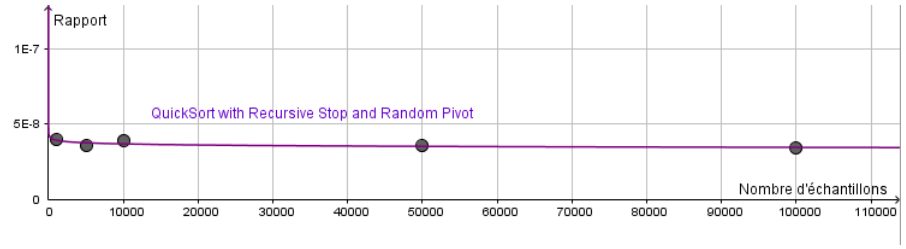

FIGURE 33 – Test du rapport en meilleur/moyen cas, quick sort à seuil et avec pivot aléatoire, exemplaires 0 à 9

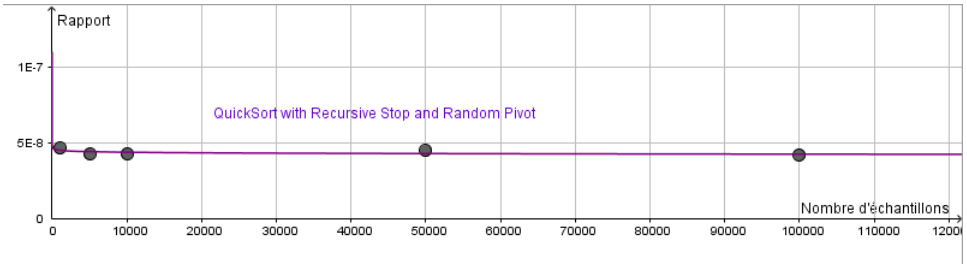

FIGURE 34 – Test du rapport en meilleur/moyen cas, quick sort à seuil et avec pivot aléatoire, exemplaires 10 à 19

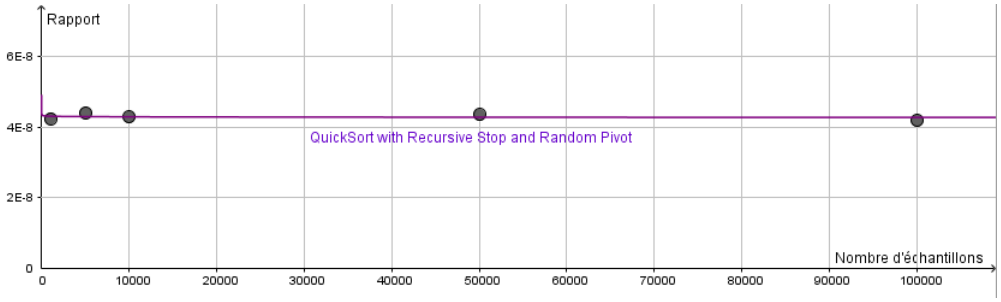

FIGURE 35 – Test du rapport en meilleur/moyen cas, quick sort à seuil et avec pivot aléatoire, exemplaires 20 à 29

(b) En pire cas :

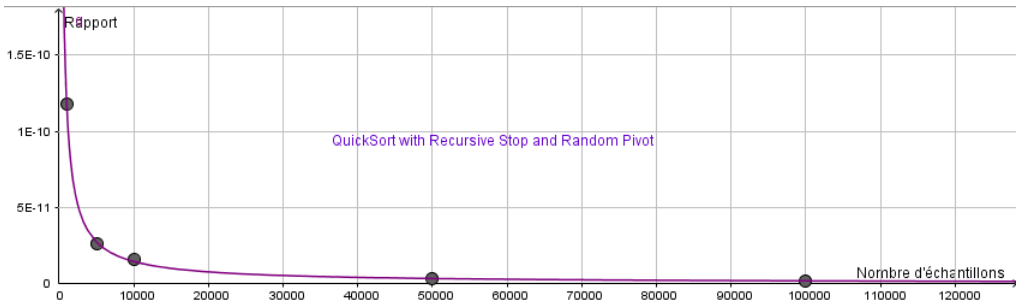

FIGURE 36 – Test du rapport en pire cas, quick sort à seuil et avec pivot aléatoire, exemplaires 0 à 9

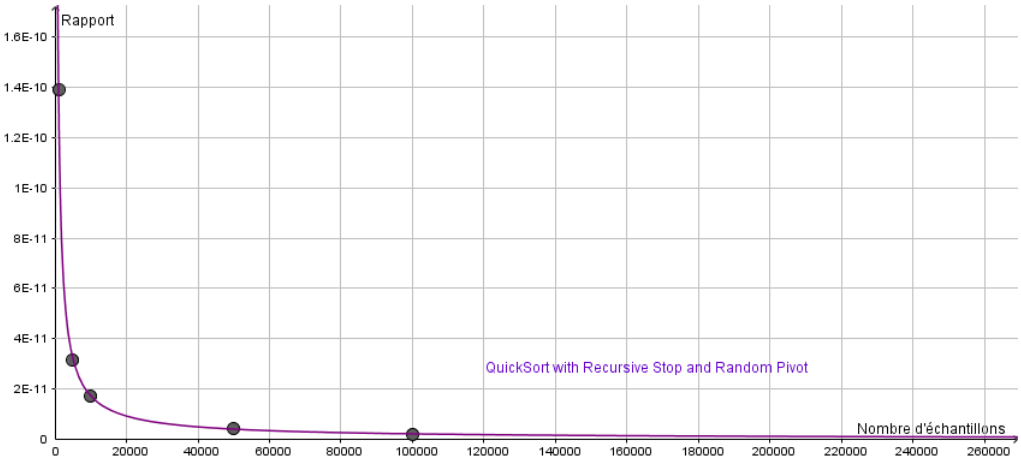

FIGURE 37 – Test du rapport en pire cas, quick sort à seuil et avec pivot aléatoire, exemplaires 10 à 19

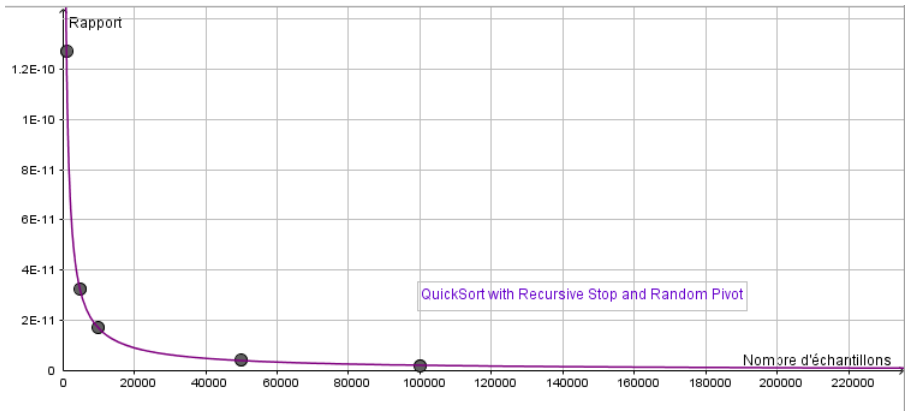

FIGURE 38 – Test du rapport en pire cas, quick sort à seuil et avec pivot aléatoire, exemplaires 20 à 29

On voit clairement que pour les quatre version du Tri Rapide et chaque série d'exemplaire, nous n'avons pas de pire cas, car les rapports en pire cas donnent une courbe tendant vers 0 ce qui veut dire qu'une complexité en $\mathcal{O}(n^2)$ est une surestimation. En revanche, le rapport en moyen et meilleur cas nous mène sur une courbe tendant vers $4 \cdot 10^{-9}$. La complexité en $\mathcal{O}(n \log n)$ semble être correcte avec une constante multiplicative de $4 \cdot 10^{-9}$.

L'interpolation des puissances effectuée par GeoGebra donne des valeurs légèrement différente de celles trouvées avec le test du rapport. Nous trouvons tout de même des résultats dans le même ordre de grandeur (10^{-9}).

2.5 Test des constantes

Pour chaque algorithme, nous avons utilisé le cas moyen comme fonction d'hypothèse, car elle donnaient les résultats attendus dans les tests précédents. On place donc les couples $(f(x), y)$ sur le graphique en espérant afficher une droite qui nous donnera la constante multiplicative via sa pente, et un coup fixe grâce à son décalage à l'origine.

Nous obtenons les droites et leurs équations suivantes :

— Pour le tri à bulles :

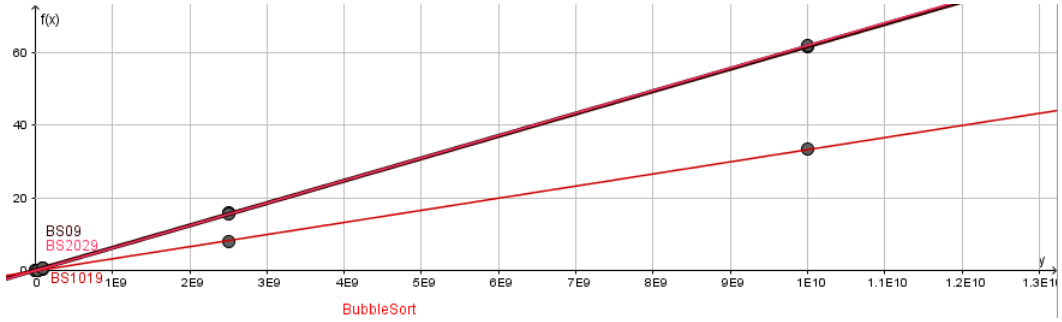

FIGURE 39 – Test des constantes, bubble sort

BS09(x) = 0.000000006149874 x + 0.124757072841219
BS1019(x) = 0.00000000333099 x − 0.078817893736151
BS2029(x) = 0.000000006189596 x + 0.008409011173043

FIGURE 40 – Valeurs du test des constantes, bubble sort

— Pour le tri rapide :

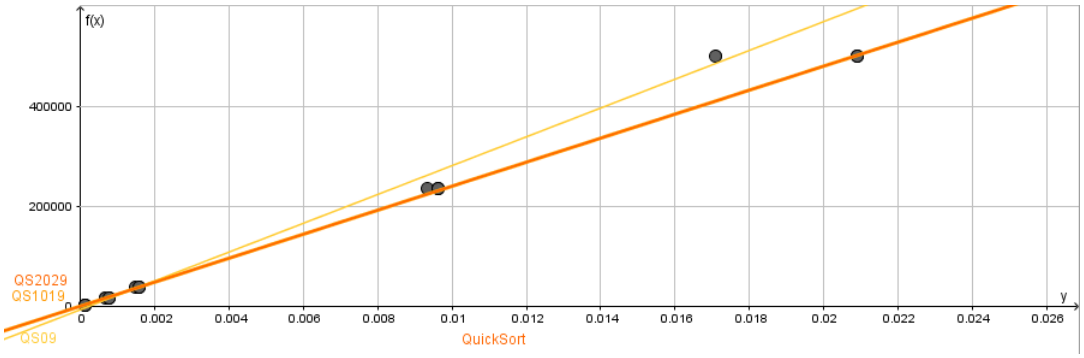

FIGURE 41 – Test des constantes, quick sort

$QS09(x) = 0.000000034633385\,x + 0.000221494111211$
$QS1019(x) = 0.000000041607672\,x + 0.000100369306168$
$QS2029(x) = 0.000000041607672\,x + 0.000100369306168$

FIGURE 42 – Valeurs du test des constantes, quick sort

— Pour le tri rapide avec pivot aléatoire

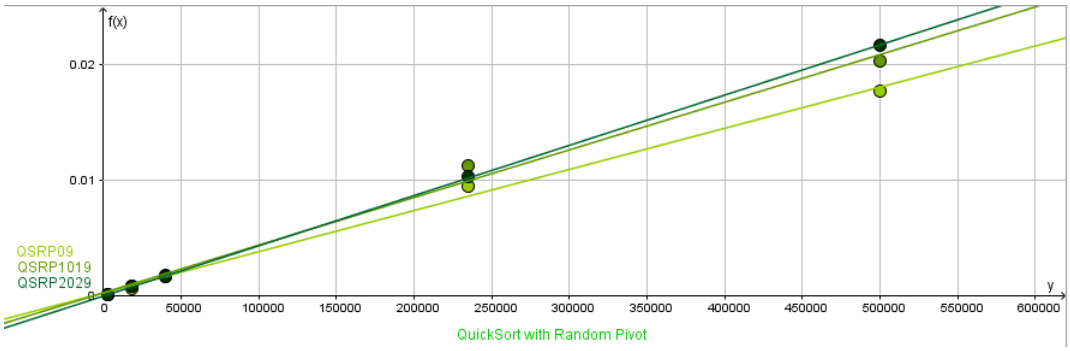

FIGURE 43 – Test des constantes, quick sort avec pivot aléatoire

$QSRP09(x) = 0.000000035579716\,x + 0.000288754381658$
$QSRP1019(x) = 0.000000041202698\,x + 0.000293277005152$
$QSRP2029(x) = 0.000000043442074\,x + 0.000013169777883$

FIGURE 44 – Valeurs du test des constantes, quick sort avec pivot aléatoire

— Pour le tri rapide avec seuil de récursivité :

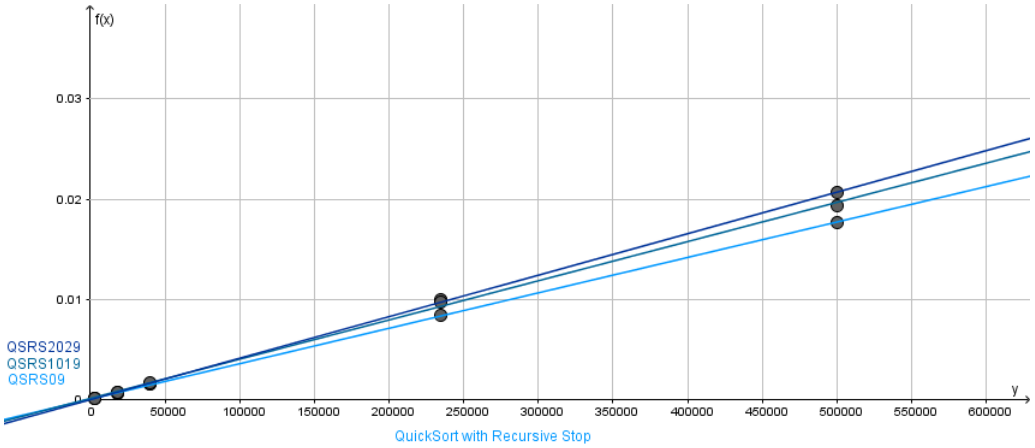

FIGURE 45 – Test des constantes, quick sort avec seuil

$QSRS09(x) = 0.000000035305804\,x + 0.000084385462525$
$QSRS1019(x) = 0.000000039080776\,x + 0.000155075216164$
$QSRS2029(x) = 0.000000041373452\,x + 0.000025877861719$

FIGURE 46 – Valeurs du test des constantes, quick sort avec seuil

— Pour le tri rapide avec pivot aléatoire et seuil de récursivité :

Nous retrouvons grâce à ce test la même constante multiplicative de chaque algorithme que dans les deux tests précédent. Cependant nous trouvons ici un coup fixe en plus qui n'avait pas

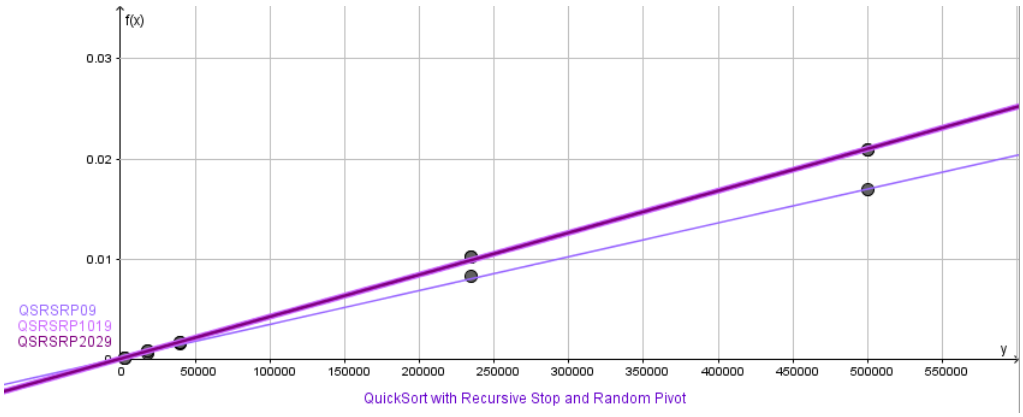

FIGURE 47 – Test des constantes, quick sort avec seuil et pivot aléatoire

$$QSRSRP09(x) = 0.000000033767593\,x + 0.000123605097154$$
$$QSRSRP1019(x) = 0.000000041902196\,x + 0.000075255005976$$
$$QSRSRP2029(x) = 0.000000041902196\,x + 0.000075255005976$$

FIGURE 48 – Valeurs du test des constantes, quick sort avec seuil et pivot aléatoire

été déterminé auparavant. Il est de l'ordre de 10^{-4} pour les algorithmes de tri rapide et assez variable pour le tri à bulle.

2.6 Quel algorithme utiliser ?

Comme on a pu le constater, l'algorithme à appliquer va beaucoup dépendre de l'exemplaire qui est passé en paramètre. Par exemple, dans le cas d'un tableau déjà trié, l'algorithme du tri à bulle est linéaire. Alors que pour le tri rapide, il est quadratique.

C'est pour cette raison que la constante optimale du seuil de recursivité varie beaucoup d'un exemplaire à un autre. Il est même parfois plus efficace de ne pas utiliser ce seuil. De même, pour certains exemplaires, il est plus intéressant de prendre le premier élément en tant que pivot plutôt qu'un élément au hasard.

Il faut donc savoir s'adapter en fonction de la situation. L'idéal serait de connaître le type d'exemplaire que l'on va nous soumettre à l'avance.

Conclusion

Grâce à ce premier TP, nous avons pu remarquer l'importance d'une analyse hybride.

En effet, avec une approche purement théorique, nous n'aurions pas été en mesure de distinguer quel tri appliquer et dans quelle situation. Nous n'aurions pas pu, non plus, percevoir les critères distinctifs de chaque amélioration de tris : on peut constater l'impact d'un changement seulement grâce à une approche empirique lorsque les ordres de grandeurs sont les mêmes.

Avec une approche purement empirique, nous aurions passé plus de temps à déterminer l'allure de la courbe avant de lui appliquer les tests. Mais surtout, nous n'aurions pas pu déterminer si cette courbe est valable pour TOUS les exemplaires du problème de tri.

En combinant ces deux approches, nous avons été en mesure de gagner du temps et nous pouvons maintenant déterminer quel tri appliquer pour un exemplaire donné.