Compte-rendu de TP

Complexité algorithmique

Aurélien SCHILTZ

6 janvier 2017

Résumé

L'objectif de ce TP est de comprendre la notion de complexité algorithmique, et de mesurer la complexité des algorithmes suivants :
— recherche du minimum dans un tableau ;
— algorithmes de tri (tri par sélection, tri par bulles, tri fusion, tri rapide) ;
— recherche binaire (dichotomie).

Note relative au code écrit pour ce TP

L'intégralité du code écrit pour ce TP est disponible sur le dépôt Github suivant, dans le dossier nommé `tp1` :

`https://github.com/aurelienshz/ii.2415-algo-prog`

Table des matières

Première partie

Méthodologie

1 Principe général

— Génération des jeux de données
— Appel des fonctions à mesurer et stockage des résultats dans des Hashmaps
— Tracé des graphiques

2 Génération de données d'entrée de longueur variable

La génération est effectuée une seule fois par la méthode `generateRandomDatasets` de la classe `ComplexityMeasurements`. Ces jeux de données sont stockés sous forme d'un tableau à deux dimensions, qui est un attribut privé de la classe ComplexityMeasurements.

Notre objectif est de travailler avec le même jeu de donnée pour chacun des algorithmes sur lesquels nous souhaitons effectuer des mesures de complexité temporelle. Afin de ne pas altérer le jeu de données initial, la classe `ArrayUtils` est instanciée pour chacun de ces algorithmes, et le jeu de données de test est cloné à chaque instanciation, puis passé en argument au constructeur de cette classe.

3 Mesure de la complexité temporelle

Pour mesurer la complexité temporelle, la classe `TimerUtils` sert à gérer le chronométrage de l'exécution d'une fonction. Elle permet de mémoriser le temps CPU du thread au sein lequel le programme s'exécute, et de récupérer la différence entre l'appel à `startTimer` et l'appel à `stopTimer`, via un appel à `getDuration`.

4 Utilisation des références de fonction

Pour appeler les fonctions que nous souhaitons mettre à l'épreuve sur notre jeu de données sans répéter le code servant à créer la HashMap qui stockera les résultats des mesures, une fonctionnalité de Java 8 est utilisée : il s'agit de la classe Consumer, qui permet de passer à une fonction dédiée à la mesure une référence vers une autre fonction, qu'elle se chargera d'appeller.

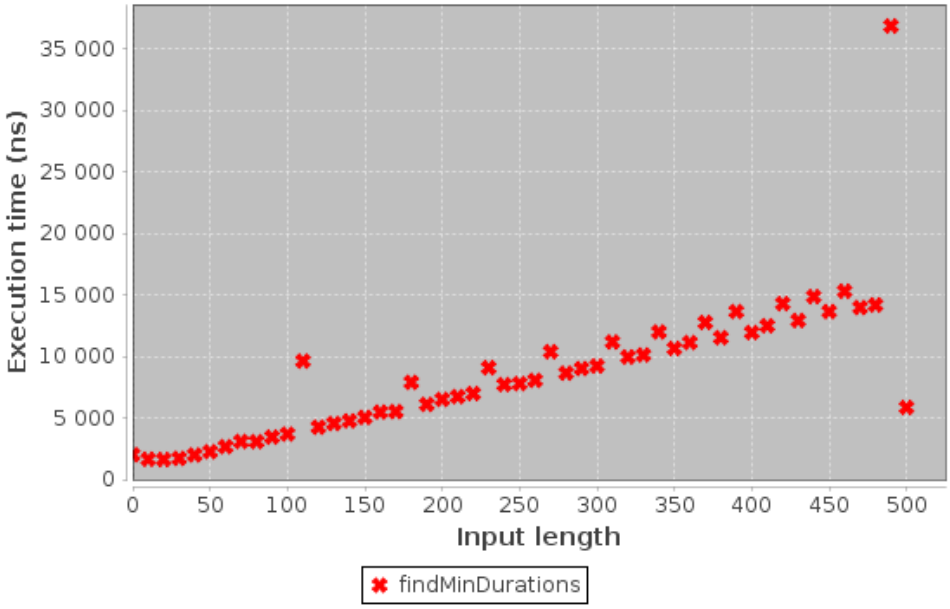
5 Tracé de graphiques

Pour tracer des graphiques, la bibliothèque JFreeChart est utilisée. Une classe a été développée pour s'interfacer avec cette bibliothèque de manière plus légère, dans la mesure où les configurations sont assez redondantes entre tous les graphiques qui doivent être tracés. Cette classe permet de tracer, dans une nouvelle fenêtre, les graphiques correspondant aux valeurs contenues dans une HashMap, grâce à sa méthode `drawHashmap`, ou de tracer sur un même graphe les données contenues dans plusieurs HashMaps grâce à sa méthode `addHashmapToDataset`, qui permet d'ajouter une nouvelle série de valeurs au graphique en cours de construction.

6 Un premier exemple : la recherche du minimum d'un tableau

Pour éprouver notre architecture, nous lançons notre programme sur la fonction servant à trouver la valeur minimale contenue dans un tableau. Cette méthode est ArrayUtils : :findMin. Elle effectue une unique itération sur le tableau, quelle que soit la position de la valeur minimale. Sa complexité temporelle théorique est donc linéaire dans tous les cas, c'est-à-dire de la forme $\mathcal{O}(n)$.

La courbe obtenue pour la mesure expérimentale de sa complexité temporelle est conforme à ce résultat théorique : elle nous montre effectivement une série de points apparentée à une fonction linéaire.



Deuxième partie

Algorithmes de tri

1 Tri par sélection

1.1 Complexité temporelle théorique

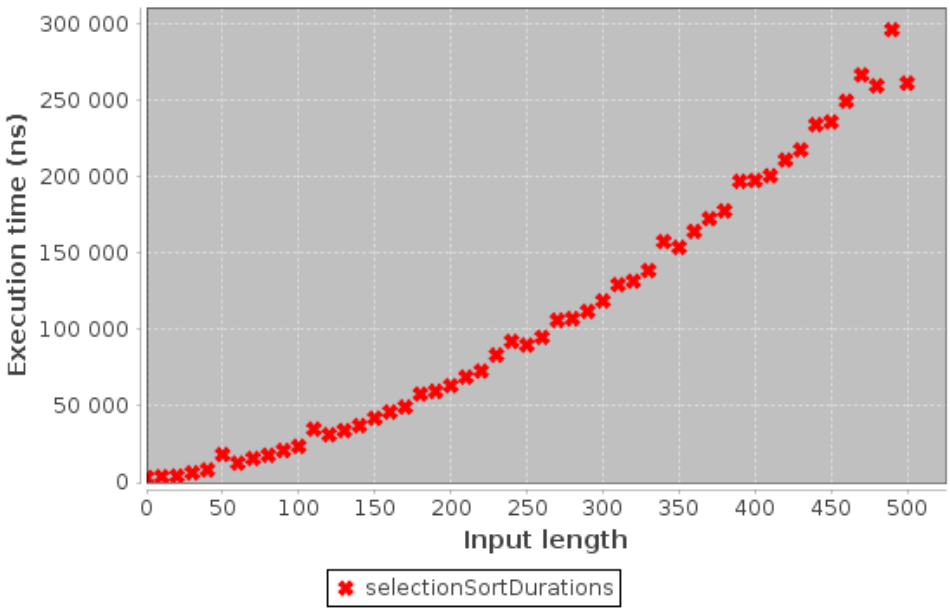
Le tri par sélection itère sur le tableau lors de l'appel à sa méthode principale (`SelectionSorter::sort`), mais parcourt intégralement la partie de tableau pas encore triée afin de trouver l'index de l'élément le plus faible pour chacune de ces parties. Autrement dit, le nombre de comparaisons (et potentiellement d'échange, suivant le résultat de cette comparaison) est :

$$\begin{aligned} N &= n + (n-1) + (n-2) + ... + (n-(n-2)) + (n-(n-1)) \\ &= \sum_{i=1}^{n} i \\ &= \frac{n(n-1))}{2} \\ &= \mathcal{O}(n^2) \end{aligned}$$

On obtient donc une complexité asymptotique en $\mathcal{O}(n^2)$, c'est-à-dire une **complexité quadratique**.

1.2 Mesure expérimentale de la complexité temporelle

L'exécution de notre programme visant à mesurer la complexité temporelle du tri par sélection nous donne le graphe suivant :



Ce graphe est cohérent avec le résultat obtenu précedemment, puisque l'on y observe une courbe de forme quadratique.

2 Tri à bulles

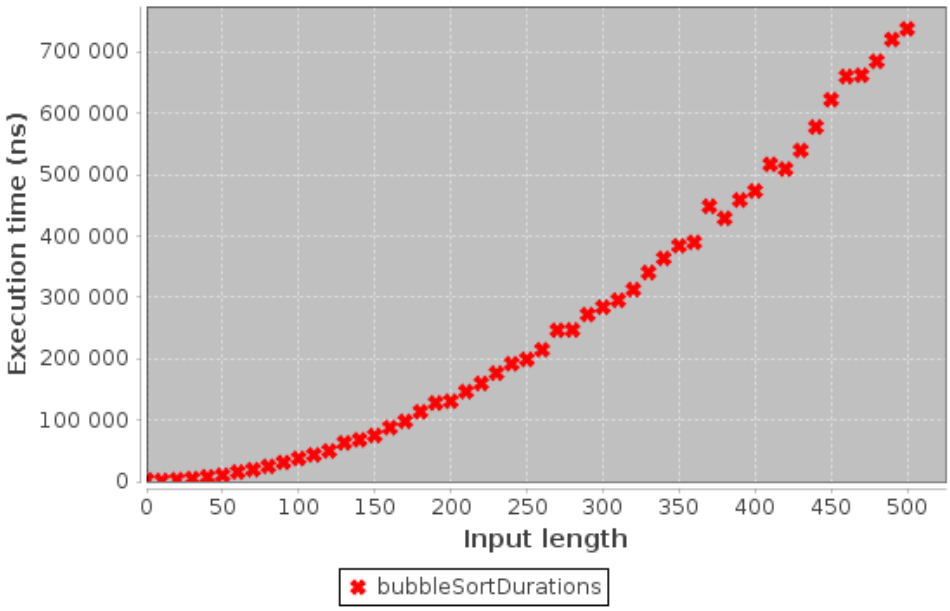
2.1 Complexité temporelle théorique

Le nombre d'itérations réalisées par le tri à bulles dépend de l'ordre initial des éléments du tableau. Dans le meilleur cas (tableau déjà trié), l'algorithme n'effectue qu'un seul parcours du tableau, et a donc une complexité linéaire. Dans le pire cas, il le parcourt autant de fois qu'il a besoin de permuter les éléments du tableau, et réalise n^2 permutations. Il a alors une complexité quadratique.

Cependant, sa complexité moyenne est également quadratique. En effet, le nombre d'itérations sur le tableau dépend du nombre de permutations nécessaire, et on peut démontrer que le nombre de permutations nécessaires pour un tableau initial trié dans un ordre aléatoire est en moyenne égal à $n(n-1)/4$.

2.2 Mesure expérimentale de la complexité temporelle

L'exécution de notre programme dessine le graphe suivant pour le tri à bulles :



Ce graphe est cohérent avec le résultat obtenu précedemment, puisque l'on y observe une courbe de forme quadratique.

3 Tri fusion

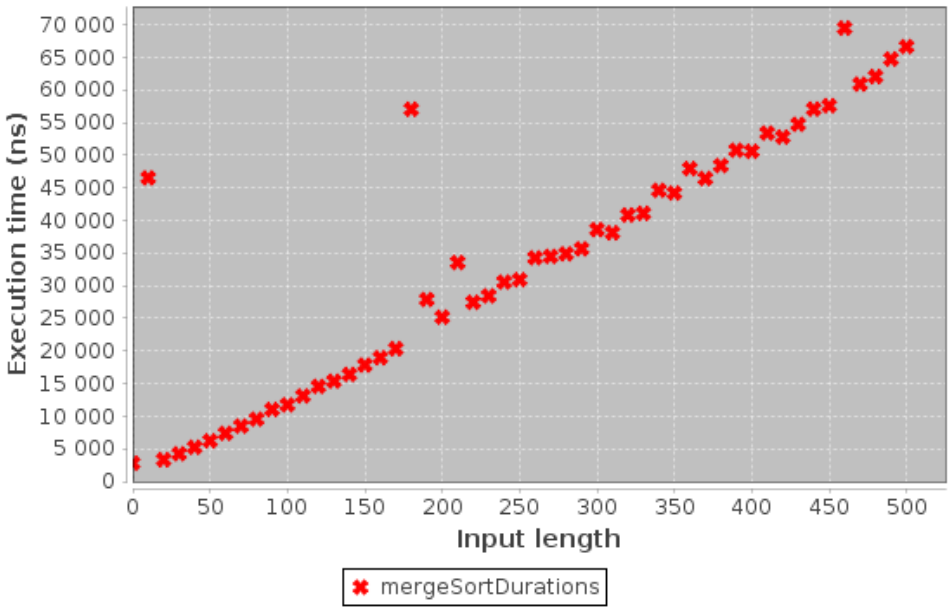
3.1 Complexité temporelle théorique

Intéressons-nous tout d'abord à la fonction servant à fusionner deux sous-tableaux triées. Celle-ci possède une complexité en $\mathcal{O}(m+n)$, où m et n sont les tailles respectives des deux tableaux. En effet, quoi qu'il arrive, elle parcourra la totalité de chacun de ces deux tableaux, puisqu'elle doit comparer la première valeur des deux tableaux pour décider laquelle insérer dans le tableau final. Il est à noter que l'implémentation proposée a une complexité spatiale minimale : elle ne réalise une copie que de la première des deux parties de tableaux qu'elle doit fusionner, et travaille directement sur le tableau final. Autrement dit, si m et n sont les tailles de ces deux tableaux, la mémoire supplémentaire qu'elle nécessite pour travailler est uniquement celle occupée par un tableau de taille m.

Puis, étudions la complexité temporelle de l'algorithme complet. Ici, on devrait trouver $n \log(n)$, mais c'est un résultat que je ne suis pas parvenu à retrouver de moi-même.

3.2 Mesure expérimentale de la complexité temporelle

L'exécution de notre programme dessine le graphe suivant pour le tri fusion :



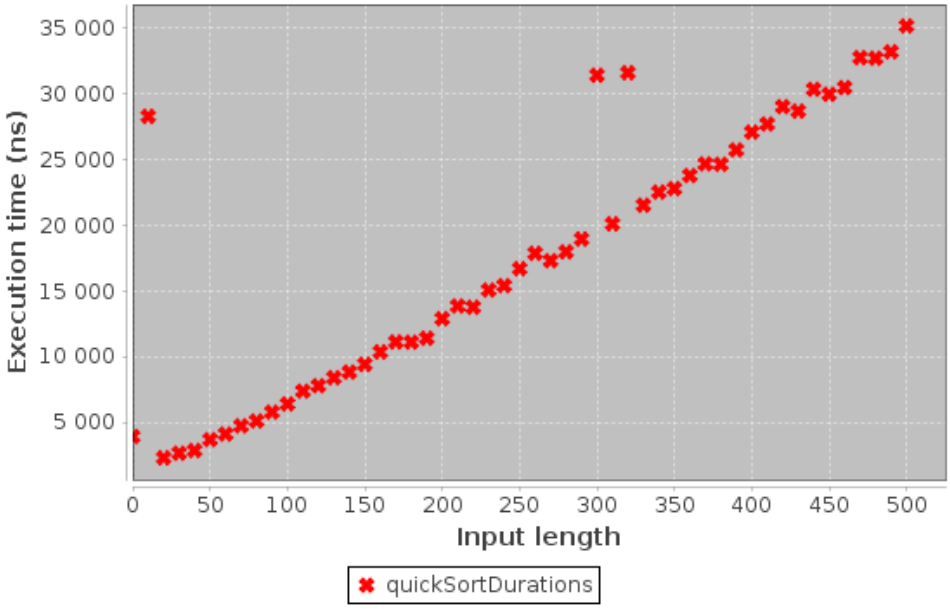
Ce graphe est cohérent avec une complexité temporelle en $n \log(n)$. En effet, la courbe représentative de la fonction $x \mapsto x \log(x)$ est très proche de celle d'une fonction linéaire, et nous observons effectivement ce type de courbe dans le graphique ci-dessus.

4 Tri rapide

4.1 Complexité temporelle théorique

4.2 Mesure expérimentale de la complexité temporelle

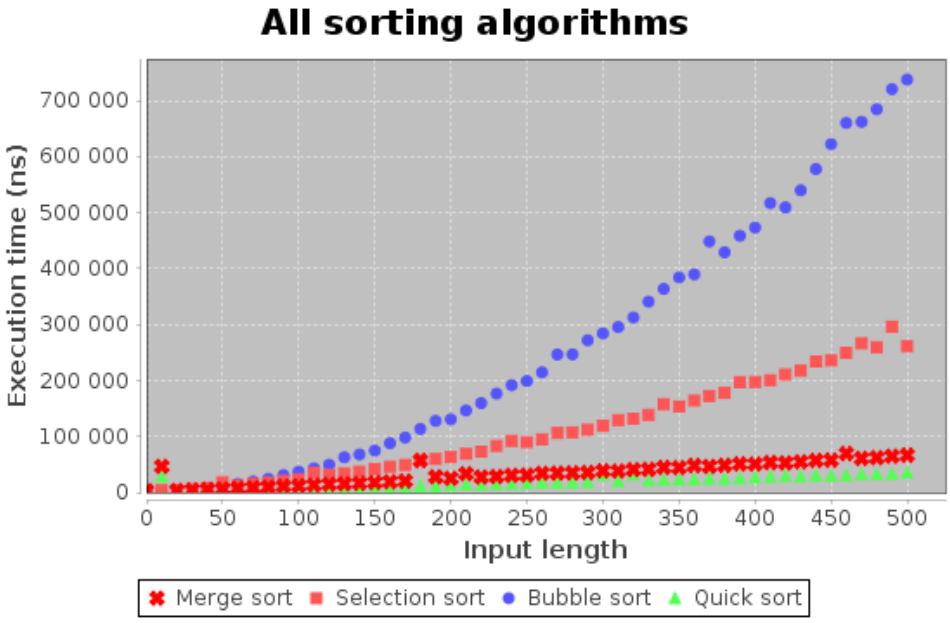
L'exécution de notre programme dessine le graphe suivant pour le tri rapide :



Nous retrouvons ici encore un graphe cohérent avec la complexité en $n \log(n)$ du tri rapide.

5 Comparaison des différents algorithmes de tri

Un aspect intéressant de l'étude de la complexité temporelle des différents algorithmes est la comparaison de leurs performances. Le tracé de leurs temps d'exécution dans un même graphique produit le résultat suivant :



Nous observons plusieurs détails intéressants sur ce graphique. Tout d'abord, on observe la large dispersion des courbes pour des tableaux de taille élevée.

Nous avons observé que le tri par sélection et le tri par bulle possèdent tous deux une complexité quadratique. Cependant, le tri par sélection réalise $n(n-1)/2$ itérations dans **tous** les cas, alors que le tri par bulle réalise en moyenne $n(n-1)/4$ itérations. On observe une courbe pour le tri par bulle qui prend des valeurs approximativement égales à la moitié de celles prises par la courbe représentant les temps d'exécution du tri par sélection, ce qui est parfaitement normal.

Enfin, le tri fusion et le tri rapide possèdent tous deux une complexité en $n \log(n)$, et ont donc des courbes très proches. Cependant, le tri rapide réalise toutes ses opérations en place dans le tableau initial, alors que le tri fusion nécessite de copier en mémoire une des deux zones à fusionner. Cette opération de copie peut expliquer le léger écart entre les deux courbes, en faveur du tri rapide.

Troisième partie

Recherche binaire

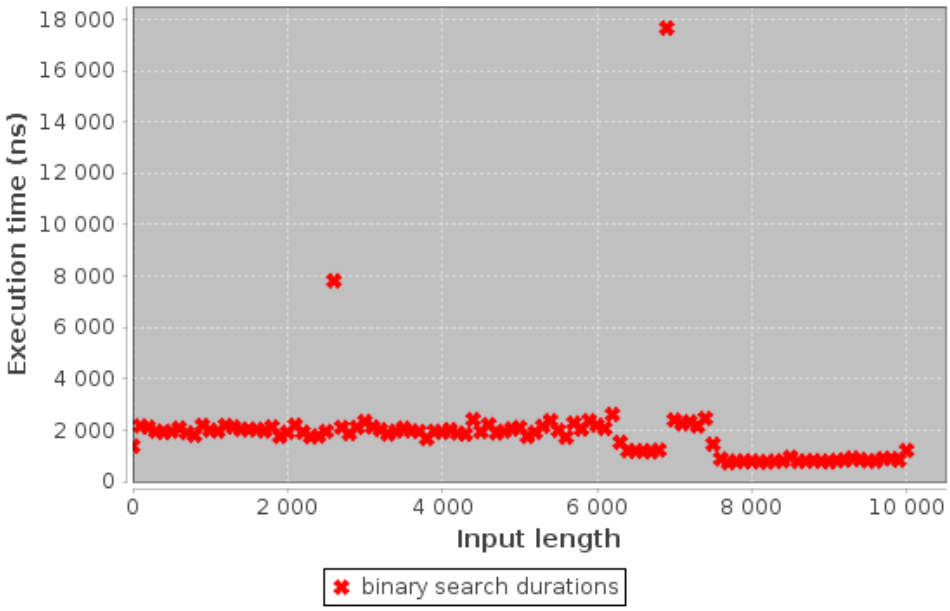
L'objectif de cette partie est de mesurer la complexité algorithmique d'une recherche par dichotomie.

1 Paramètres expérimentaux

Les paramètres ont été changé par rapport à toutes les autres mesures : nous avons pris des tableau de valeurs de tailles alant de 0 à 10000, échelonnées par palier de 100 valeurs. Afin d'apporter de la répétabilité à nos mesures, nous avons constamment recherché l'index du chiffre zéro dans les tableaux générés.

2 Résultat expérimental

Le résultat obtenu après avoir lancé notre programme est le graphique suivant :



Ce graphique est difficilement exploitable. Par ailleurs, plusieurs tentatives ont été menées pour faire varier les paramètres expérimentaux, et se sont toutes révéles infructueuses.