****

**TCO**

**Laboratoire 1**

Numa Trezzini

1. Introduction 3

2. But 3

3. Détermination de la complexité 3

3.1 Fonctionnement des algorithmes 3

3.1.1 Généralités sur les algorithmes de tri 3

3.1.2 L’algorithme de tri rapide (QuickSort) 4

3.2 Mise à l’épreuve des algorithmes 4

3.3 Résultats 5

4. Analyse des résultats 6

# Introduction

L’un des aspects majeurs de l’efficacité d’utilisation d’un programme est sa rapidité. L’ordinateur n’étant pas capable des mêmes capacités de discernement que l’humain, il requiert des algorithmes de calculs précis et bien souvent plus séquentiels que la même opération effectuée par une personne. Pour décrire le nombre d’opérations que prend un ordinateur à effectuer un certain algorithme, la notion de complexité a été introduite. Il est donc possible de déterminer mathématiquement une fourchette de temps d’exécutions pour un algorithme, en fonction du nombre d’éléments à traiter. Bien sur, une telle tache n’est pas simple. La nature de l’algorithme, premièrement, détermine la complexité : plus une structure est parcourue, plus le temps de traitement sera long. Il convient donc de limiter au maximum l’interaction avec les données à traiter. Cette interaction est toujours bornée inférieurement, car il est impossible pour une machine de « deviner » une réponse. Les données doivent ainsi toutes être traitées.

Déterminer la complexité d’un algorithme peut se faire de plusieurs façons : il est possible d’éstimer statistiquement la fonction limitante, avec des mesures de temps. Il est également possible d’étudier l’algorithme, et déduire de son fonctionnement le nombre d’opérations à effectuer.

# 2. But

L’objectif de ce laboratoire est d’explorer empiriquement la complexité de l’algorithme de tri quick sort. Différentes méthodes pour réaliser ceci seront utilisées : une régression linéaire effectuée à partir de temps de tris mesurés et le paramétrage d’une fonction de complexité représentant au plus proche les temps théoriques et pratiques.

Pour effectuer ces mesures, nous allons utiliser deux algorithmes de Quick Sort : le qsort du langage C et le q\_sort fourni avec la donnée du laboratoire[[1]](#footnote-1). Des détails sur l’algorithme sont donnés dans la section 3.

# 3. Détermination de la complexité

L’objectif de cette section est de comprendre les algorithmes fournis et de déterminer leur complexité. Nous allons d’abord nous attarder sur le fonctionnement des algorithmes, puis présenter le test de mesure des temps, en enfin déterminer la complexité à partir des mesures.

## 3.1 Fonctionnement des algorithmes

### 3.1.1 Généralités sur les algorithmes de tri

Dans le cas d’un algorithme de tri, nous pouvons dire, de façon naive, que chaque donnée doit etre comparée avec toutes les autres afin de déterminer sa place dans le « classement ». Il est possible d’effectuer ceci avec une double boucle. la complexité ainsi déterminée est de *n2*. Il est cependant possible, et meme souhaitable, d’avoir comme borne inférieure *nlog(n)* opérations, dans le cas moyen[[2]](#footnote-2). Il est possible de réaliser ceci notamment en partitionnant les données et en traitant celles-ci comme étant un nouveau problème. Les *n* données doivent être examinées au moins une fois et en moyenne *log(n)* d’entre elles sont traitées. Le pire des cas se présente lorsque les données vont à l’encontre de l’algorithme (plus de détails à la section) : il a besoin de plus d’opérations que de coutume pour résoudre le problème. On parle dans ce cas de *complexité dans le pire des cas* (*worst case*). Dans le cas d’un algorithme de tri, cette complexité dans le pire des cas est, de manière générale, *n2*. Certains algorithmes ont toutefois réussi à garder une complexité dans le pire des cas de *nlog(n)*, ce qui n’est pas le cas du QuickSort comme nous le verrons plus loin.

### 3.1.2 L’algorithme de tri rapide (QuickSort)

Le principe de l’algorithme de tri rapide consiste à définir une valeur, appelée pivot, et d’effectuer un premier tri consistant à déplacer toutes les valeurs plus petites que le pivot dans la « partie gauche » du tableau et toutes les valeurs plus grandes « à droite ». Une fois ce prétraitement effectué, les tableaux des deux cotés du pivot sont triés selon le même principe.

Le choix du pivot est crucial dans cet algorithme, comme nous le verrons plus loin (section 3.2).

#### 3.1.2.1 qsort (proposé dans une librairie C standard)

Une première solution consiste à systematiquement choisir le pivot au milieu du tableau. C’est la solution adoptée par le qsort offert par C. Le pivot est donc choisi au milieu du tableau puis déplacé tout au début. Le reste du tableau est ensuite parcouru, une fois par l’avant et une fois par l’arrière. A chaque fois qu’une valeur plus grande, respectivement plus petite est rencontrée, l’algorithme les échange. Une fois cette opération terminée, le pivot reprend sa place entre les tableaux, il est donc à sa place dans le tri. Les indices des sous-tableaux résultants sont stockés, puis traités tour à tour de la même manière que précédemment. L’ordre de tri des sous-tableaux est du type *depth-first*. C’est-à-dire que, dans la mesure du possible, à chaque apparition d’un sous-tableau, celui-ci est tout de suite traité.

#### 3.1.2.2 q\_sort (proposé par le proffesseur)

Une autre solution, celle proposée dans l’algorithme fourni par le professeur, consiste à prendre comme pivot la première valeur du tableau. L’algorithme parcourt ensuite le tableau depuis la fin jusqu'à trouver une valeur plus petite que le pivot. Cette valeur est mise à la place du pivot si elle fait partie de la seconde moitié du tableau. L’algorithme parcourt ensuite le tableau depuis le pivot jusqu'à trouver une valeur plus grande que celui-ci. Si la valeur est dans la première moitié du tableau, elle est envoyée à la place d’une valeur dans la deuxième moitié du tableau. Tant que tout le tableau n’a pas été parcouru, ces opérations continuent. A la fin de ces opérations, le pivot est remis en place dans le tableau à la place du dernier élément modifié. Les tableaux séparés par le pivot sont triés à leur tour. A ce moment, le sous-tableau de gauche ne contient que des valeurs plus petites que le pivot, et le sous-tableau droit que des valeurs plus grandes.

## 3.2 Mise à l’épreuve des algorithmes

Pour tester l’efficacité des algorithmes de tri, nous allons mesurer les temps de tri pour des tableaux de double répartis uniformément entre 0 et 1. Ces valeurs seront d’abord aléatoires puis prédéfinies afin de vérifier les limites imposées par l’algorithme. Cette dernière façon de faire permettra de faire ressortir le comportement dans le pire des cas.

Dans le cas de l’algorithme qsort de C, les données présentant le pire des cas sont celles dont le pivot est systématiquement la valeur la plus basse ou haute du tableau, sachant que le pivot est toujours choisi au milieu du tableau.

Dans le cas de l’algorithme q\_sort fourni, les données présentant le pire des cas sont celles triées en ordre inverse (décroissant, ici) de celui recherché. Les comparaisons et échanges explosent dans cette configuration, car le pivot, toujours le premier élément du tableau, est systématiquement le plus grand du tableau.

Comme nous pouvons le constater avec la définition des données dans le pire des cas, le pivot joue un rôle central dans la durée d’exécution du quicksort. En effet, plus le pivot est proche de la médiane des valeurs du tableau, moins le nombre de comparaisons et d’échanges sera grand. On pourra ainsi plus facilement séparer les tableaux en deux autres de longueur à peu près égale, respectant ainsi la complexité moyenne en *O(nlog(n))* du quicksort. Dans le pire des cas, lorsqu’il n’y a qu’un seul sous-tableau à chaque itération, l’algorithme triera donc un seul élément à la fois (le pivot).

Afin de mesurer les temps de tri, nous avons choisi de marquer le résultat fourni par la fonction *clock()* avant le début et après la fin du tri. La différence entre ces deux valeurs donne le nombre de microsecondes écoulées entre le début et la fin du tri. Pour finir le calcul du temps, nous avons transformé le résultat obtenu en secondes. Une division par 106 produit le résultat escompté.

## 3.3 Résultats

Les résultats produits par les tests sont présentés ci-dessous. Les temps sont donnés en secondes.

Les résultats obtenus lors du tri avec des données aléatoires sont les suivantes :

|  |  |  |
| --- | --- | --- |
| Nombre d’éléments | temps qsort | temps q\_sort |
| 10 | 0.000007 | 0.000003 |
| 30 | 0.000007 | 0.000004 |
| 100 | 0.000019 | 0.000014 |
| 300 | 0.000061 | 0.000046 |
| 1000 | 0.000202 | 0.000171 |
| 3000 | 0.000656 | 0.000571 |
| 10000 | 0.002469 | 0.002158 |
| 30000 | 0.008237 | 0.007092 |
| 100000 | 0.031005 | 0.025962 |
| 300000 | 0.100133 | 0.08505 |
| 1000000 | 0.365682 | 0.299323 |
| 3000000 | 1.187781 | 0.966376 |
| 10000000 | 4.251702 | 3.476204 |
| 30000000 | 14.129258 | 12.043444 |

Le tri avec des données triées dans l’ordre décroissant donne les résultats suivants :

|  |  |  |
| --- | --- | --- |
| nb\_elements | temps qsort | temps q\_sort |
| 10 | 0.000005 | 0.000002 |
| 30 | 0.000005 | 0.000004 |
| 100 | 0.000013 | 0.000025 |
| 300 | 0.000032 | 0.000202 |
| 1000 | 0.000096 | 0.002123 |
| 3000 | 0.000281 | 0.0197 |
| 10000 | 0.000994 | 0.242309 |
| 30000 | 0.003444 | 2.292255 |
| 100000 | 0.009207 | 25.093002 |

Les résultats avec le qsort qui part en vrille :

# 4. Analyse des résultats

## 4.1 Résultats pour des données réparties aléatoirement

Figure - graphe log-log d'une répartition aléatoire des données

Figure - graphe log-normal d'une répartition aléatoire des données

## 4.2 Résultats pour des données désavantageant le q\_sort

Figure - graphe log-log d'une répartition des données défavorisant le q\_sort

Figure - graphe log-normal d'une répartition des données défavorisant le q\_sort

## 4.3 Résultats pour des données désavantageant le qsort

# 5. Conclusion

1. Adaptation d’un code de <http://www.personal.kent.edu/~rmuhamma> [↑](#footnote-ref-1)
2. Un tableau déjà trié demande souvent *n* opérations, vu qu’il s’agit juste de lire les données pour constater qu’elles sont déjà triées [↑](#footnote-ref-2)