Rapport Projet N°1

Rapport portant sur le cours *Technique de Programmation*

Document rédigé par **David Taralla**
2e Bachelier en Sciences Informatiques

Dernière version : 9 mars 2011

Table des matières

Partie 1

Présentation du One-way Stackless Quicksort

Cette partie explique par la méthode des décompositions successives l'algorithme de Huang Bing-Chao et Donald E. Knuth nommé *One-way Stackless Quicksort*. Après avoir présenté le code de la fonction de tri, je commencerai par expliquer l'évolution du marqueur `i` durant l'algorithme et le fonctionnement du cœur de la boucle principale (imbriquée dans une autre et malheureusement implémentée au moyen d'un `goto`). C'est le premier fragment de la fonction complète, un autre l'englobant et l'utilisant pour trier le tableau fourni en entrée.

Ce fragment « parent » est ensuite analysé, mais plus rapidement car son fonctionnement se base sur ce qui a été introduit en première section.

1.1 Implémentation C du *Quicksort One-way*

```
static void quicksort(int* tab, int low, int high) {
    int i, j;
    int K;

    while (low < high) {
        if (tab[low] < 0) {
            tab[low] = -tab[low];
            low++;
        }
        else {
            i = j = low;
            K = tab[low];
            inc: j++;

            while (K <= tab[j]) j++;
```

```
            if (tab[j] > 0) {
                tab[i] = tab[j];
                i++;
                tab[j] = tab[i];

                goto inc; // Horrible, mais nécessaire
            }

            tab[i] = -K;
        }
    }
}
```

1.2 Évolution du marqueur *i* et analyse du cœur de la boucle

Au départ, `i` vaut `low` qui vaut 0. `i` va servir à déterminer la position finale de `K`. On incrémente `i` seulement lorsque l'on a trouvé un `tab[j]` positif et strictement supérieur à `K`.

Dans le corps du `if` (ledit cœur de boucle, notre « opération atomique »), on remarque que l'on dédouble à chaque fois un élément : `tab[i]` devient `tab[j]` et `tab[j]` devient `tab[i+1]`. Et voici le rôle du marqueur `i` : enregistrer la position de l'origine du dernier doublon créé de cette manière.

Le marqueur `j` quant à lui représente la position à laquelle on va dédoubler ce qui est à l'index `i`, si ce qui se trouve à l'index `j` n'est pas déjà bien placé (c'est-à-dire, s'il n'a pas encore été « marqué » d'un signe négatif) évidemment.

Avant de regarder si un dédoublage doit avoir lieu, le marqueur `j` est modifié. On l'incrémente une première fois (car cela ne servirait à rien de vérifier qu'un élément est bien trié par rapport à lui même). Ensuite, tant que notre élément `K` est inférieur ou égal à l'élément d'indice `j`, cela signifie que `K` est déjà bien placé par rapport à cet élément et ceux qui le précèdent, on incrémente donc `j`.

Ainsi, en fin de parcours, lorsque le gardien du bloc `if` devient faux parce que `j` ne peut plus être incrémenté et `tab[j]` est négatif, `i` pointe par construction vers la position finale de `K`.

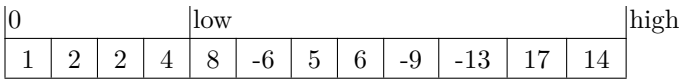
On place alors la valeur `-K` à cet index, et on retrouve une permutation du tableau de départ (si on ignore les signes), sans doublons.

1.3 Évolution du marqueur *low* : Construction de la partie triée finale

Dans cette grande boucle contenant la principale, l'invariant est le suivant : à tout moment, on a (voir illustration Figure 1.1)

– `tab[0:low-1]` est la partie triée finale du tableau. Les entiers qu'elle contient sont tous strictement positifs.
– `tab[low:high-1]` contient des éléments en cours de tri (non triés ; strictement positifs) et des éléments dont la position finale est déjà déterminée (strictement négatifs).

Figure 1.1 – Illustration de l'invariant



Ainsi, lorsque `low` devient égal à `high`, cela signifie que la partie triée est égale au tableau tout entier. Mais quand et pourquoi le marqueur `low` est-il incrémenté ?

Après chaque fin de la boucle principale (celle dont j'ai expliqué l'utilité en Section 1.2), on regarde si l'élément `tab[low]` est négatif ou positif. S'il est positif, cela veut dire que cet élément n'est pas encore trié par rapport au reste du tableau. On recommence donc la boucle principale afin de le placer à sa place finale et correcte.

Par contre, si cet élément est négatif, on peut prendre l'opposé de l'élément `tab[low]` et incrémenter `low` tout en conservant l'invariant. C'est ce qui est fait.

Ainsi, au fur et à mesure de l'exécution de cette grande boucle, `low` va avancer, toujours de gauche à droite [1], jusqu'à devenir égal à `high`.

Plus formellement, on sait que cette boucle s'arrêtera toujours car une fonction de terminaison de celle-ci est donnée par

$$\begin{aligned} T &= 3N - nbPivotsPlacés - 2 \times nbElementsTriés \\ &= 3N - nbPivotsPlacés - 2 \times low \end{aligned}$$

Ceci termine mon étude par décompositions successives du *One-way, Stackless Quicksort*.

1. D'où la dénomination « One-way » de cette version de *Quicksort*

Partie 2

Comparaison des performances

Je présente dans cette partie un comparatif des performances du *One-way, Stackless Quicksort* face à celles du *Three-way Quicksort*. J'y exprime la méthodologie adoptée avant de présenter les résultats que j'ai obtenus.

2.1 Méthodologie

J'ai testé les deux algorithmes séparément sur une machine de l'institut Montefiore. Je l'ai fait pour plusieurs tailles de listes, les entiers présents dans ces listes variant de 1 à 999 (compris) dans un premier temps, puis variant de 1 à 99 999 (compris) dans un second temps.

Enfin, je termine par une étude du comportement des deux algorithmes lorsque des tableaux entièrement triés leur sont fournis en entrée (toujours pour les mêmes tailles).

Le caractère étudié est le temps d'exécution. Celui-ci variant fort d'une machine à l'autre, ce qui est important c'est bien les différences nettes, les *rapports* entre les temps observés sur l'exécution des deux algorithmes.

J'ai utilisé la librairie *time.h* et sa fonction `clock()`, en l'appelant avant (t_0) et après (t_1) l'exécution de chacune des version du *Quicksort*.[1]

Les temps que j'ai mesurés étaient les résultats de chaque calcul $(t_1 - t_0)$/`CLOCKS_PER_SEC` et sont précis au centième de seconde (le système UNIX ne peut être plus précis et le langage `C` ne permet pas de faire mieux).

2.2 Analyse et présentation des mesures

J'ai effectué dix tests avec des graines aléatoires différentes pour cinq tailles de tableaux, puis ai pris la moyenne des temps de tri, c'est pourquoi les mesures

1. Voir *README* et fichiers joints

peuvent paraître plus précises que le centième de seconde. Ce sont ces moyennes que vous verrez dans les tableaux 2.1 et 2.2.

Première expérience La première observation a été faite en donnant en entrée aux deux algorithmes des tableaux d'entiers compris entre 1 et 999. Cet intervalle est assez restreint et c'est voulu. Les résultats obtenus sont présentés dans le tableau 2.1.

Table 2.1 – Mesures pour listes d'entiers compris entre 1 et 999

	Algorithmes				
Tailles		**100 000**	**500 000**	**1 000 000**	**1 500 000**
	`QSort 1-way`	0.023	0.325	1.169	2.535
	`QSort 3-way`	0.014	0.071	0.141	0.212

On remarque qu'ici, le *Quicksort One-way* est un assez lent comparé à son pendant *Three-way*. Cela peut s'expliquer par le phénomène suivant. Les tableaux étant de très grandes tailles, et les entiers qu'ils contiennent ne variant que de 1 à 999, les tableaux formés aléatoirement contiennent beaucoup de redondances. Le *One-way* ne gère pas ces dernières de façon spéciale et elles le ralentissent, alors que le *Three-way* prend parti de ces redondances pour ne pas trier ce qui est égal à l'élément pivot dans la fonction de partition.
D'où une situation qui avantage le *Quicksort Three-way*.

Deuxième expérience Désormais, les listes contiennent des entiers ne variant plus de 1 à 999 mais bien de 1 à 99 999, ce qui amène des résultats bien plus raisonnables pour le *Quicksort One-way* (voir Tableau 2.2).

Table 2.2 – Mesures pour listes d'entiers compris entre 1 et 99 999

	Algorithmes				
Tailles		**100 000**	**500 000**	**1 000 000**	**1 500 000**
	`QSort 1-way`	0.019	0.100	0.209	0.323
	`QSort 3-way`	0.029	0.133	0.256	0.388

On note désormais une meilleure performance du *One-way*. Avec des tableaux contenant des redondances moindres, ce dernier effectue un meilleur score que le *Three-way* ! Je conclus des deux dernières expériences que, en règle générale, le *Quicksort One-way* est plus rapide que l'autre. Il est cependant dépassé par le *Three-way* si le tableau fourni en entrée est tel qu'il avantage ce dernier (c'est à dire, avec des redondances).

Troisième expérience Un dernier test de performances peut être effectué pour montrer que le *Quicksort One-way* est plus « rapide » et réagit « mieux » (toutes proportions gardées...) que le *Quicksort Three-way* en règle générale.

Entrons, toujours pour des mêmes tailles, des tableaux déjà entièrement triés. Les résultats sont présentés dans le tableau 2.3.

Note Ici, un seul test a été lancé pour chaque taille vu qu'il n'y a pas d'aléatoire, et les tests ne portent que sur les quatre premières tailles car c'est suffisant pour illustrer la comparaison.

Table 2.3 – Mesures pour listes d'entiers triés, de 1 à [taille].

	Algorithmes			
Tailles		**100 000**	**500 000**	**1 000 000**
	`QSort 1-way`	10.71	271.01	1114.51
	`QSort 3-way`	43.99	N/A	N/A

Ici, le *Quicksort Three-way* a complètement dégénéré. De plus, utilisant la pile, sa récursion va si loin que les machines de l'institut Montefiore s'arrêtent après approximativement 5-10 minutes de calculs avec une erreur de segmentation.

Le caractère `stackless` du *One-way* permet d'éviter cette erreur. De plus, comme à son habitude avec des tableaux non redondants, cette version de l'algorithme est plus performante que l'autre, même si elle souffre aussi beaucoup de l'entrée défavorable.

2.3 Conclusion tirée des résultats

En conclusion, on remarque que la version *One-way, stackless* du *Quicksort* est plus performante et plus robuste que sa version *Three-way*. Elle dégénère aussi si l'entrée est un tableau déjà entièrement trié, mais moins.

Ensuite, on note que le *Quicksort One-way*, face à une situation particulière avantageant le *Quicksort Three-way* peut paraître moins performant (bien qu'en moyenne, dans une situation « normale », il n'en est rien).

À plusieurs points de vue donc, sur des tableaux de grande taille en tout cas, l'utilisation de cet algorithme sera préconisée si on a besoin de gagner quelques centièmes de seconde dans les cas généraux.

Remarque Je suis conscient que l'analyse des performances est plus longue que prévue avec cette mise en page. Néanmoins, si on réduit les marges et l'espace des titres, on arrive facilement à moins de deux pages de contenu...