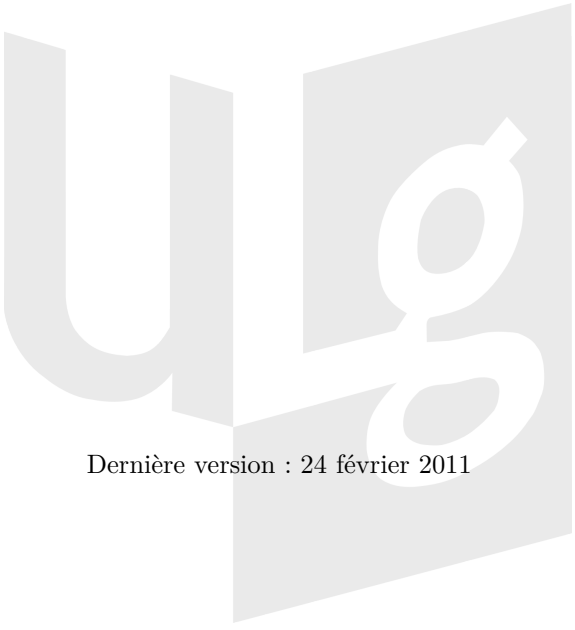
M'enfin, ce Quicksort !

Synthèse portant sur le cours *Technique de Programmation*

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



Dernière version : 24 février 2011

Préambule

Ce document est le fruit d'une après-midi de travail à comprendre l'algorithme *Quicksort*. Il présente les implémentations `C` de sa fonction de partitionnement (2-way), avant d'expliquer cette dernière en détail. La fonction `quicksort()` est ensuite présentée.

Table des matières

Partie 1

La fonction de partitionnement

1.1 Description de la fonction

Cette fonction s'articule autour d'une boucle `while` principale. C'est cette dernière qui s'occupe de faire tout le travail ; les instructions situées en fin de fonction placent le pivot à la bonne position et retournent la position finale et correcte de ce pivot. Notons `ref` l'indice maximal de la liste (ici, on choisit le pivot en fin de tableau).

Précondition

On considère que l'on passe à la fonction les arguments suivants :
- `tab` pointe vers un tableau non vide d'entiers.
- `low` est un entier supérieur ou égal à 0.
- `high` est un entier strictement inférieur à `ref - 1`.

Invariant de boucle

À chaque tour de boucle, l'invariant suivant est respecté :
- `tab[0:low-1]` contient des entiers inférieurs ou égaux à la valeur du pivot. Notons cette partie du tableau `A`.
- `tab[low:high]` contient des entiers qui n'ont pas encore été analysés et qui doivent encore probablement être replacés. Notons cette partie du tableau `B`.
- `tab[high+1:ref-1]` contient des entiers strictement supérieurs à la valeur du pivot. Notons cette partie du tableau `C`.

Postcondition

La partie `B` du tableau est vide et `low` contient l'indice de la place correcte du pivot.

1.2 Implémentation C

```
static int partition(int* tab, int low, int high) {
    int ref = high--;

    while (low <= high) {
        if (tab[low] <= tab[ref])
            low++;
        else {
            int tmp   = tab[high];
            tab[high] = tab[low];
            tab[low]  = tmp;
            high--;
        }
    }

    int tmp  = tab[ref];
    tab[ref] = tab[low];
    tab[low] = tmp;
    return low;
}
```

1.3 Complexité

La complexité est clairement *O(n)* (linéaire). En effet, dans le pire des cas on switchera $n-1$ éléments (cas où tous les éléments sont strictement supérieurs à `tab[ref]` par exemple).

1.4 Explication complète

La zone indéterminée `B` est réduite à chaque tour de boucle, que ce soit par la droite (décrément du marqueur `high`) ou par la gauche (incrément du marqueur `low`). À chaque tour, on regarde si `tab[low]` est plus petit ou égal à la référence `tab[ref]`.

S'il est inférieur, on n'y touche pas et on incrémente `low` (en effet, vu l'invariant, cela veut dire que l'élément est à sa place : on élargit la partie `A` et on réduit `B`).

S'il est supérieur, on le switche avec l'élément à l'indice `high` (d'ailleurs, on ne sait pas ce que celui-ci vaut), on décrémente ensuite `high` sans toucher à

`low` qui au prochain tour de boucle, permettra d'évaluer cet élément switché inconnu et le placer au bon endroit. On sait que l'invariant reste respecté : `C` est élargie alors que B est réduite, et l'élément est bien positionné par rapport au reste selon l'invariant.

À la fin de l'algorithme, à la sortie de la boucle, on place l'élément de référence au bon endroit (on le switche avec `tab[low]`).

Mais pourquoi à l'indice `low` ?

Parce que, que l'on termine la boucle par un `high--` ou un `low++`, `low` sera l'indice du premier élément de `tab` qui est strictement plus grand que `tab[ref]`.

En effet, lors de l'avant-dernière itération, on a `high - low = 1`. Ensuite, qu'on exécute `high--` ou `low++`, on arrive à `low = high`. Le dernier tour de boucle compare donc `tab[low == high]` avec lui-même, on incrémente donc `low` une dernière fois.

Désormais, `high - low = -1`, la boucle s'arrête, et on a plus qu'à échanger `tab[ref]` avec `tab[low]` pour placer `tab[ref]` au bon endroit.

La postcondition est respectée. À savoir, on a désormais 2 parties (dont l'une peut être vide) séparées par l'élément de référence : avant lui, les éléments ont des valeurs qui lui sont inférieures ou égales, et après lui, les valeurs lui sont strictement supérieures.

Partie 2

La fonction de tri

2.1 Description de la fonction

Le mécanisme central de cette fonction est la récursion. *Quicksort* est un algorithme de type "Diviser pour régner".

Diviser

On procède à une *division* du tableau en deux sous-tableaux (éventuellement vides) `tab[p:q-1]` et `tab[q+1:r]` tels que chaque élément de `tab[p:q-1]` soit inférieur ou égal à `tab[q]` qui, lui-même, est inférieur ou égal à chaque élément de `tab[q+1:r]`.

L'indice `q` est obtenu grâce à la valeur de retour de la fonction de partitionnement.

Régner

Les deux sous-tableaux `tab[p:q-1]` et `tab[q+1:r]` sont triés par des appels récursifs au tri rapide.

2.2 Implémentation C

```
static void quicksort(int* tab, int low, int high) {
    if (low < high) {
        int q = partition(tab, low, high);
        quicksort(tab, low, q-1);
        quicksort(tab, q, high);
    }
}
```

Partie 3

Annexes

3.1 Améliorations possibles

1. Techniques alternatives du choix du pivot
   - Pivot aléatoire : résout *en moyenne* la dégénérescence ; il n'est désormais plus possible de provoquer une dégénérescence.
   - Pivot central (*mauvaise technique*).
   - Pivot *median-of-three*.
   - Pivot *median-of-median-of-three*.
2. Combinaison avec d'autres algorithmes de tri
   - À partir d'un certain temps d'exécution, passer au *Heapsort*.
   - Lorsque les zones à trier deviennent "petites" (environ 15 éléments), le *tri par insertion* devient plus intéressant, plus efficace (point de vue temps comme point de vue espace).
   - Pour le tri de chaines de caractères, le partitionnement en *drapeau hollandais* (aussi nommé *3-way Quicksort*) est plus intéressant.