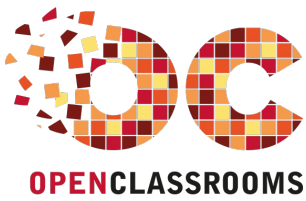
Le tri rapide : QSort

Par GuilOooo



www.openclassrooms.com

Dernière mise à jour le 19/07/2009

Sommaire

QSort Le tri rapide : QSort

Par  GuilOooo

Mise à jour : 19/07/2009

Salut à tous !
Aujourd'hui nous allons apprendre une nouvelle sorte de tri : le tri rapide, aussi appelé QSort (*QuickSort* = tri rapide en anglais). Le principe est assez original et donne de bons résultats sur les listes très désordonnées. Par contre, sur une liste presque déjà triée, il faudra trouver autre chose...

Dans tous les cas, je vous propose de découvrir ce nouvel algorithme sans plus tarder !
Sommaire du tutoriel :

QSort

Le principe

Bon, alors cet algorithme a un principe un peu original si vous êtes habitués au tri à bulles ou au tri par sélection. En fait, l'idée, c'est de séparer votre tableau en deux. Pour cela, on choisit une valeur de notre tableau de base, qu'on appelle **pivot**. Le pivot est souvent la valeur de la première case du tableau. On construit alors deux « sous-tableaux » : l'un contient toutes les valeurs du premier tableau qui sont **inférieures ou égales** au pivot, l'autre contient les valeurs **supérieures** au pivot.

Oui, mais mon tableau n'est pas trié, là ?

Non, il ne l'est pas. Mais il est déjà un peu plus « rangé ». Afin de terminer le tri, il vous suffira de trier à nouveau chacun des deux sous-tableaux avec `qsort`, puis de **concaténer** les deux tableaux (cela signifie les mettre bout à bout, comme `strcat` le fait avec les chaînes de caractères en C).

Comme vous pouvez le voir, l'algorithme du tri rapide s'utilise lui-même. On dit que c'est un algorithme récursif (bluestorm a écrit un tutoriel sur la récursivité, pour ceux que ça intéresse). Le dernier problème est donc de savoir quand on arrête d'appeler récursivement QSort... La réponse est très simple : quand il n'y a plus rien à trier (c'est-à-dire, quand les sous-tableaux restants sont soit des singletons, soit vides) !

Un petit exemple ?

Prenons cette liste :

Code : Autre

```
42 5 38 37 21
```

Ici, je vais choisir comme pivot 37. En effet, il y a deux valeurs au-dessus (38 et 42) et deux valeurs au-dessous (5 et 21). C'est donc une valeur médiane, ce qui convient très bien pour un pivot (on verra que le principal problème de l'implémentation de QuickSort est le choix du pivot). On prend donc la case 37 comme pivot et on fait passer de l'autre côté du pivot les valeurs qui

ne sont pas à leur place :

Code : Autre

```
21 5 37 42 38
-------------
<37  || >37
```

Ici, on voit la liste initiale séparée en deux : au milieu le pivot, 37. À gauche, les valeurs inférieures à 37 (<37) ont été insérées. De même à droite pour les valeurs supérieures à 37 (>37).

Bon, maintenant on est sûr que le 37 est à sa position définitive : il ne bougera plus. En effet, toutes les valeurs qui lui sont inférieures sont à sa gauche, et les autres à droite.

Vous pouvez donc recommencer à appliquer le tri rapide pour les deux sous-listes restantes : (21, 5) et (42, 38). Cependant, dites-vous bien que dans un certain nombre de cas, quand on arrive à des sous-listes de cette taille, elles sont déjà « naturellement » triées (ceci dépend de l'implémentation). Pour tout vous dire, j'en ai bavé pour trouver un exemple de liste qui ne soit pas triée immédiatement.

Code : Autre

```
21  5      * 37 *     42 38
---------------------------
<21 || >21 ****** <42 || >42
```

Ce schéma montre les deux QuickSort appliqués chacun à une des sous-listes. On voit ici ces deux QuickSort effectués simultanément, juste après le choix des pivots, et avant le déplacement des valeurs de part et d'autre de ceux-ci.

Comme vous le voyez, on trie les deux listes complètement indépendamment. Bon, ici je me force à choisir un pivot et tout, mais il existe des variantes de QSort qui, à un certain stade, changent d'algorithme pour trier les sous-listes qui restent.
Une fois qu'on a fini le tri des deux sous-listes restantes, on obtient donc ceci :

Code : Autre

```
5   21     * 37 *    38 42
---------------------------
<21 || >21 ****** <42 || >42
```

Les deux QuickSort précédents, après l'étape de déplacement des valeurs. On remarque que les groupes de nombres entre les pivots sont soit des singletons, soit vides. QuickSort est donc terminé.

Nous sommes sûrs que les nombres 21, 37 et 38 sont à leur place définitive. En l'occurrence, il se trouve que la liste est triée. En effet, les sous-listes qui restent ne font qu'un nombre de long. On arrête donc de trier récursivement...

Voilà : vous avez saisi l'astuce ?

Oui, mais ton algorithme va impliquer de créer plein de sous-tableaux, et de sous-sous-tableaux, de les parcourir... T'es sûr que ça va être si rapide que ça ? Et comment on va bien pouvoir l'implémenter ?

Il existe une implémentation du tri rapide qui est en place (c'est-à-dire qu'elle n'utilise aucune mémoire supplémentaire, en dehors de celle du tableau initial). Elle élimine donc pas mal de problèmes d'utilisation de mémoire et de puissance de calcul.

Cependant, nous verrons d'abord comment implémenter une version « simple » de QSort. Tout ça se passe dans la sous-partie suivante... En avant !

Passons à la pratique !

Bon allez, implémentons tout ça.

QSort simple en OCaml

En premier lieu, nous allons implémenter un QSort qui obéit à la lettre à la définition de la sous-partie 1. Comme on l'a vu, cette implémentation risque d'être assez lourde à gérer à cause de tous les sous-tableaux. Pourtant, il est possible d'exprimer QuickSort de façon très simple dans certains langages. Observons par exemple cette implémentation en OCaml par bluestorm (si vous ne connaissez rien à OCaml, voyez par là, là, et sur Google).

Code : OCaml

```
let rec qsort liste = match liste with
  [] | _::[] -> liste
| pivot::reste ->
    let debut, fin = List.partition (fun a -> a < pivot) reste in
    qsort debut @ [pivot] @ qsort fin
```

Ce code commence par déclarer la fonction **qsort**, récursive, qui prend un paramètre (**liste**), et qui regarde quelle est la structure de la liste (c'est le **match liste with**). Il y a alors plusieurs cas possibles :

- si c'est une liste vide ou à un seul élément, alors on retourne la liste elle-même ;
- sinon, on appelle **pivot** le 1er élément de la liste, et **reste** le reste ;
  puis on partitionne le **reste** en deux listes, **début** et **fin**, en utilisant l'expression **a < pivot** pour décider dans quelle liste placer chaque élément **a** ;
  et enfin, on met bout à bout (c'est l'opérateur **@**) **debut**, le **pivot**, et **fin**, en appelant **qsort** sur **debut** et **fin** avant.

On voit donc que l'algorithme du tri rapide s'exprime de manière très simple et concise en OCaml. Ce code est sans doute « traduisible » en C, mais la partie « partition de la liste » risque d'être bien lourde à écrire. Il doit sûrement exister une autre solution pour les langages impératifs...

Une autre implémentation, en C

L'idée est extrêmement simple. Plutôt que de créer un nouveau tableau à chaque fois pour appeler récursivement **qsort** dessus, nous allons travailler sur un seul et unique tableau : celui qu'on nous fournit au départ.

Comment faire ?

Voici l'astuce : nous allons faire une fonction **quickSort()** qui prend trois paramètres, comme ceci :

Code : C

```
void quickSort(int tableau[], int debut, int fin);
```

Le premier est bien évidemment le tableau à trier. Quant au second et au troisième, ce sont des indices (c'est-à-dire des numéros de cases). La fonction **quickSort** se chargera donc de trier le tableau entre la case numéro **debut** et la case numéro **fin** ! Pour appeler récursivement **quickSort** par la suite, il suffira de la rappeler avec le même tableau et les **debut**/**fin** qui vont bien. Plus besoin de créer des sous-tableaux !

Le seul obstacle qu'il nous reste : comment se débrouiller pour faire passer toutes les valeurs inférieures au pivot à sa gauche, et toutes les valeurs supérieures à sa droite ? Déjà, comme pour la version Caml, nous choisirons comme pivot le premier élément du tableau, pour simplifier.

Ensuite, nous allons faire deux **for**. L'un d'entre eux parcourra le tableau de droite à gauche à la recherche d'un élément inférieur au pivot, (qui n'est donc pas à sa place) ; l'autre parcourra le tableau de gauche à droite à la recherche d'un élément supérieur au pivot.

Dès que les deux **for** ont chacun trouvé un élément, on permute ces deux éléments. Puis on recommence, jusqu'à ce que les **for** se « croisent » (l'indice du premier **for** est supérieur à celui du second). À ce moment, le tableau est prêt : l'ensemble des valeurs inférieures au pivot sont situées avant les valeurs supérieures au pivot. Il ne nous reste plus qu'à trier chacune de ces deux parties indépendamment, avec QuickSort !

Compliqué ? Je vous l'accorde. Voici une implémentation. Je vous recommande de la faire tourner avec différents tableaux, en y ajoutant des **printf** un peu partout pour comprendre ce qui se passe.

Code : C

```
void echanger(int tableau[], int a, int b)
{
    int temp = tableau[a];
    tableau[a] = tableau[b];
    tableau[b] = temp;
}

void quickSort(int tableau[], int debut, int fin)
{
    int gauche = debut-1;
    int droite = fin+1;
    const int pivot = tableau[debut];

    /* Si le tableau est de longueur nulle, il n'y a rien à faire.
*/
    if(debut >= fin)
        return;

    /* Sinon, on parcourt le tableau, une fois de droite à gauche,
et une
autre de gauche à droite, à la recherche d'éléments mal placés,
que l'on permute. Si les deux parcours se croisent, on arrête. */
    while(1)
    {
        do droite--; while(tableau[droite] > pivot);
        do gauche++; while(tableau[gauche] < pivot);

        if(gauche < droite)
            echanger(tableau, gauche, droite);
        else break;
    }

    /* Maintenant, tous les éléments inférieurs au pivot sont avant
ceux
supérieurs au pivot. On a donc deux groupes de cases à trier. On
utilise
pour cela... la méthode quickSort elle-même ! */
    quickSort(tableau, debut, droite);
    quickSort(tableau, droite+1, fin);
}
```

Merci à Shareman pour ses améliorations sur ce code.

Que vous pouvez appeler comme dans cet exemple :

Code : C

```
int main(void)
{
    int tab[5] = {5, 3, 4, 1, 2};
    int i;
```

```
    quickSort(tab, 0, 4);

    for(i = 0; i < 5; i++)
    {
 printf("%d ", tab[i]);
    }
    putchar('\n');

    return 0;
}
```

Bonne lecture et bonne prise de tête !

Attention, en langage C il existe une fonction standard appelée « **qsort** », qui est déclarée dans stdlib.h.
Évitez donc d'appeler votre fonction de tri « qsort », car cela pourrait entraîner des collisions.

Dans la dernière partie, nous verrons les performances et optimisations possibles de QuickSort...

Rapide... À quel point ?

Cet algorithme s'appelle le « tri rapide ». Alors *rapide, rapide,* c'est vite dit. Dans quelle mesure est-il rapide ?

Nous allons ici regarder la complexité de l'algorithme.
En d'autres termes, nous allons chercher à évaluer le nombre d'opérations nécessaires au tri de notre tableau en fonction de la taille de ce dernier. Cette évaluation ne sera pas précise : nous ne dirons pas « il va falloir 1032 opérations pour ranger un tableau de 700 cases ». Nous allons plutôt chercher à voir comment le temps d'exécution varie lorsque la taille du tableau elle-même varie.

Cependant, cette valeur n'est pas toujours la même, elle dépend naturellement des données que nous devons trier. Nous allons donc étudier deux cas : le pire cas possible, et le meilleur cas possible.

Pire cas possible

Prenons un exemple simple, le cas dans lequel quickSort est le moins bon. Supposons que l'on veuille ranger :

10 9 8 7 6 5 4 3 2 1

avec notre algorithme. Ça vous paraît ridicule ? Pourtant, quickSort ne peut pas « voir » qu'il suffit d'inverser les valeurs, et il va trier comme d'habitude. On choisit donc la première valeur, **10**, comme pivot, puis on effectue la partition :

1 9 8 7 6 5 4 3 2 **10**

Note : les valeurs en gras sont celles qui sont placées définitivement, et qui seront donc ignorées par l'algorithme dans la suite de son déroulement.

On a simplement permuté **1** et **10**. Mais toutes les autres valeurs sont inférieures à **10**, donc aucune n'est déplacée ! Maintenant, on recommence la même opération, en ignorant le **10** bien placé à la fin. On choisit **1** comme pivot. Or, toutes les valeurs lui étant supérieures sont après lui dans le tableau. On ne fait donc rien (mais on a quand même parcouru le tableau une fois pour s'en rendre compte).

Rebelotte en ignorant le **1** et le **10** :

1 2 8 7 6 5 4 3 **9 10**

On se retrouve en fait dans la même situation qu'initialement : le **9**, plus grande valeur du tableau, a été permutée avec le 2, puis, comme toutes les autres valeurs sont inférieures à 9, nos deux recherches d'éléments mal placés se sont « croisées », et on n'a rien permuté de plus.
De même, après cette étape, un autre passage déterminera que le **2** est bien placé et qu'on peut l'ignorer.

Ce schéma se répètera ainsi jusqu'à ce que tout le tableau soit trié. On se rend compte que, à chaque étape, on place correctement **une** valeur et que, pour ce faire, on parcourt tout l'ensemble tableau à trier. Ainsi, le premier passage fait 10 tests, le second 9, et ainsi de suite. Au final, on aura effectué 10 + 9 + 8 + ... + 3 + 2 + 1 = **55** tests et **10** permutations (une par passage).

Cette idée se généralise très facilement à des tableaux de longueur **N** : quickSort fera N + (N - 1) + (N - 2) + ... + 3 + 2 + 1 tests, et N permutations. En calculant la somme N + (N - 1) + (N - 2) + ... + 3 + 2 + 1, on trouve qu'elle est égale à **N*(N+1)/2**, soit **grossièrement N²**. Ainsi, si vous doublez la taille du tableau, dans le pire des cas, le temps d'exécution quadruplera.

Meilleur cas possible

Nous avons vu le cas pessimiste, le pire comportement possible de quickSort. Maintenant, quel est son meilleur comportement ? Prenons un nouvel exemple. Cette fois-ci, les valeurs à trier sont :

6 1 2 5 4 3 9 8 7 11 10

Ici, on choisit comme pivot le numéro **6**. On le permute avec la première valeur rencontrée (en partant de la droite) qui lui est inférieure, à savoir le 3. Il n'y a aucune autre permutation à faire, on divise donc le tableau en deux :

3 1 2 5 4 **6** 9 8 7 11 10

On trie maintenant chaque tableau indépendamment. Dans celui de droite, je choisis **3** comme pivot, et je le permute avec la première valeur qui lui est inférieure en partant de la droite : le 2. De même pour l'autre tableau, on choisit **9** comme pivot, et on le permute avec le **7**. On divise chacun des tableaux en deux et on obtient...

2 1 **3** 5 4 **6** 7 8 **9** 11 10

... un tableau tout coloré ! Une dernière étape, triviale, consiste à ordonner les différents tableaux de 2 éléments. Ce qui nous donne...

1 2 **3** 4 5 **6** 7 8 **9** 10 11

... un tableau trié.

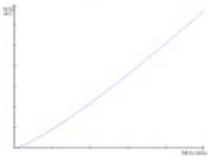
Regardons maintenant combien d'opérations nous avons eu à effectuer. Pour la première étape, nous avons simplement parcouru le tableau, nous avons donc effectué **11** opérations. À la seconde étape, nous avons parcouru deux sous-tableaux faisant chacun 5 cases, donc en tout **10** opérations. À la troisième étape, nous avons parcouru 4 tableaux de 2 cases chacun, soit **8** opérations. Soit un total de 11 + 10 + 8 = **29 tests** effectués. Pour les déplacements, nous en avons fait **1**, puis **2**, puis **4**, soit **7 permutations**.

La question se pose maintenant de savoir comment généraliser ce résultat à un tableau de longueur **N** ? Pour ce faire, nous devons d'abord remarquer qu'à chaque étape du processus, nous doublons le nombre de sous-tableaux considérés, mais que la taille de chacun de ces sous-tableaux est divisée par deux. Donc, au final, chaque étape demandera **N** opérations.

Il nous faut maintenant compter le nombre d'étapes nécessaires pour trier un tableau. Pour cela, on se pose la question : « quand dois-je arrêter de trier ? Quand le travail est-il fini ? ». Le tri est bien évidemment terminé lorsque nous atteignons des sous-tableaux de taille 1. Moralité : il y a autant d'étapes que de divisions par 2 nécessaires pour passer de N à 1. La fonction qui nous dit « combien de fois diviser N par 2 avant d'arriver à 1 » existe : c'est une fonction qui s'appelle le *logarithme de base 2*, que l'on notera ici **log**.

Du coup, le tri rapide a fait **N * log(N)** tests. Cela signifie que si je double la taille du tableau, dans le meilleur des cas, on double le temps d'exécution, et on lui ajoute une certaine durée (toujours la même).

À titre indicatif **uniquement**, voici le tracé de la courbe **N*log(N)**, de manière à ce que ceux qui ne voient pas bien à quoi elle ressemble puissent se faire une idée. J'insiste : elle est juste là pour vous donner une idée, n'essayez pas de vous en servir pour mesurer le temps d'exécution de l'algorithme. D'ailleurs, les chiffres ont été volontairement effacés.



Le nombre de permutations, lui, est moins évident à évaluer, nous ne le traiterons donc pas ici.

Dans le pire cas, on a bien vu quel tableau provoquait le comportement lent : quand le tableau est rangé à l'envers. Mais, dans le meilleur des cas, on ne voit pas bien quelles caractéristiques font que le tableau provoque un bon tri. Comment les reconnaître ?

Les tableaux provoquant les meilleurs cas sont ceux dont le pivot est proche de la **médiane** du tableau, c'est-à-dire qu'il y a autant de valeurs supérieures au pivot que de valeurs inférieures.
Mon exemple était choisi de manière à ce que chaque sous-tableau soit également un tableau optimal, il en a donc résulté que le tri rapide fut particulièrement efficace dans ce cas.

Au final...

Les cas que nous avons vus ici sont des extrêmes. Dans la vie de tous les jours, les tableaux que vous allez trier ne seront pas comme ceux que l'on a vus, ils oscilleront quelque part entre les deux. De même, l'efficacité du tri variera entre les deux extrêmes que nous avons calculés.

Cependant, comme nous avons vu quels facteurs tendaient à nous rapprocher du meilleur cas ou du pire cas, on peut tenter de jouer sur eux afin de produire souvent de bons cas. On peut par exemple choisir un pivot plus intelligent que la première valeur de la série.

L'optimisation du tri rapide est un domaine qui prendrait un tutoriel entier à vous enseigner, nous verrons donc cela dans un prochain épisode.
Voilà, vous pouvez maintenant trier vos tableaux avec QSort. C'est une méthode de tri très utilisée, vous la rencontrerez sûrement à un moment ou à un autre.

Pour plus d'informations, et de nombreuses implémentations, je vous renvoie encore une fois à la page Wikipédia qui traite du sujet.
Si vous avez un problème, une remarque, ou une suggestion, n'hésitez pas à me contacter par MP.

Bon codage.

Partager

Ce tutoriel a été corrigé par les zCorrecteurs.