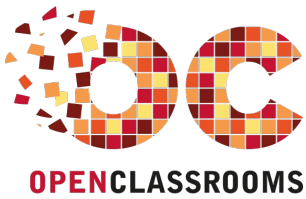
Le tri à paniers

Par GuilOooo
et Olivier Strebler (shareman)



www.openclassrooms.com

Dernière mise à jour le 13/01/2010

Sommaire

Sommaire 2
Le tri à paniers 3
Le principe 3
Implémentation du tri 5
Petits calculs de complexité 6
Partager 7

Le tri à paniers

Par  GuilOooo et  Olivier Strebler (shareman)

Mise à jour : 13/01/2010

Bonjour à tous.

Aujourd'hui je vais vous montrer un nouvel algorithme de tri. Il va nous permettre de trier un tableau de nombres **entiers** (et non réels, vous verrez pourquoi quand je vous donnerai le principe). Il est surtout utilisé lorsque vous voulez trier des entiers très proches les uns des autres, sans quoi l'algorithme commence à prendre beaucoup de place en mémoire.

L'intérêt est essentiellement « culturel », vu que la méthode employée est assez originale.

Mise à jour du 18/04/2009 : reformulation d'à peu près tout le tutoriel, plus lifting des exemple et implémentation C++ par ShareMan. Merci à lui !
Sommaire du tutoriel :

- Le principe
- Implémentation du tri
- Petits calculs de complexité

Le principe

Le principe est assez simple : on va compter le nombre d'occurences de chaque nombre de la liste, pour réécrire celle-ci. Ainsi, on va compter le nombre de « 2 », de « 3 », ... qu'elle contient, puis écrire x « 2 », suivis par y « 3 », et ainsi de suite...

Prenons un exemple. Imaginons que j'aie cette liste à trier dans l'ordre croissant :

Code : Autre

```
2 - 8 - 6 - 4 - 10 - 12 - 6 - 4
```

La première étape, est de déterminer le plus petit nombre de la liste, ainsi que le plus grand. Donc ici, ces nombres sont 2 et 12. Jusque là, c'est facile à comprendre et à implémenter .

Ensuite, nous allons créer autant de « **paniers** » qu'il n'y a de nombres entre 2 et 12. Pour connaître le nombre de paniers, on utilise la formule (max - min) + 1. Dans notre exemple, on obtient 11. On crée donc un panier par nombre possible dans la liste (compris entre le minimum et le maximum). Chaque panier correspondra ainsi au nombre d'occurrences d'un élément dans la liste : le premier panier (numéro 2) contiendra le nombre de 2 dans la liste, le second (numéro 3) le nombre de 3 dans la liste, et ainsi de suite.

Ceci nous explique pourquoi ce tri ne fonctionne pas avec les réels, ni avec d'autres éléments que les entiers : entre deux réels, il y a une infinité d'autres réels. Vu que l'on ne peut pas créer une infinité de paniers, trier des réels avec cet algorithme est donc interdit !

Revenons à notre exemple. Pour le moment on a :

Code : Autre

```
Liste à trier :

2 - 8 - 6 - 4 - 10 - 12 - 6 - 4


Paniers :

02 : 0
03 : 0
04 : 0
05 : 0
06 : 0
07 : 0
08 : 0
09 : 0
10 : 0
11 : 0
12 : 0
```

L'étape suivante est de parcourir la liste à trier, du début à la fin, et, pour chaque valeur rencontrée, de rajouter 1 « **jeton** » dans le panier qui lui correspond (c'est-à-dire incrémenter le compteur).

Code : Autre

```
Liste à trier :

2 - 8 - 6 - 4 - 10 - 12 - 6 - 4


Paniers :

02 : 1
03 : 0
04 : 2
05 : 0
06 : 2
07 : 0
08 : 1
09 : 0
10 : 1
11 : 0
12 : 1
```

Et on obtient ainsi le nombre d'occurences de chaque élément de la liste !
Maintenant, on va s'en servir pour réécrire entièrement la liste à trier. On prend le 1er panier, et on écrit sa valeur correspondante autant de fois qu'il y a de jetons dedans. On recommence avec le 2e panier, le 3e, et ainsi de suite, en écrivant toujours les valeurs à la suite.

Exemple : dans le 1er panier, le 02, il y a 1 jeton : on écrit une fois 02 dans le tableau. Le panier suivant, 03, ne contient rien. On passe. Le 04 contient 2 jetons : on écrit deux fois le nombre 04 dans le tableau. Et ainsi de suite...

On obtient alors :

Code : Autre

```
02 - 04 - 04 - 06 - 06 - 08 - 10 - 12
```

Hourrah ! La liste est triée !

Implémentation du tri

Ici, nous allons donc vous présenter une implémentation de l'algorithme du tri à paniers en C++ par ShareMan, qui permet de trier un vector (tableau dynamique). Ne soyez pas effrayés par le terme de "vector", il s'agit simplement d'une classe de la STL (la bibliothèque standard du C++) encapsulant les détails parfois complexes de la gestion et de la manipulation des tableaux dynamiques (en C, peut-être connaissez-vous malloc et free qui permettent d'utiliser ce genre de tableau ?). std::vector est LA bonne solution pour manipuler les tableaux dynamiques (et les tableaux tout courts) en C++ (il ne faut pas oublier que ce langage est orienté objet). Dans le code qui va suivre, chaque ligne, ou presque, est commentée, il ne devrait donc pas être trop dur à déchiffrer, même pour les gens qui ne font pas de C++ :

Code : C++

```
void tri_paniers(std::vector<int> & vec)
{
    /* On commence par trouver le minimum et le maximum du vector
*/
    int min = *std::min_element(vec.begin(),vec.end());
    int max = *std::max_element(vec.begin(),vec.end());

    /* Puis on crée un nouveau vector, qui représente les paniers
*/
    std::vector<int> paniers(max-min+1);

    /* Ensuite, on parcourt le tableau, et on remplit les paniers
(opération de comptage)*/
    for(unsigned int i = 0; i < vec.size(); ++i)
        paniers[vec[i]-min]++;

    /* Et enfin, on réécrit le tableau à partir de nos paniers. */
    for(unsigned int i=0, c=0; i < paniers.size(); ++i)
        for(int j=0; j < paniers[i]; ++j, ++c)
            vec[c] = i+min;
}
```

L'algorithme étant relativement simple, le code source est très court. Notez que la fonction présentée ici trie des tableaux (std::vector) d'entiers ; contrairement à une majorité d'autres algorithmes de tri (principalement ceux se basant sur les comparaisons), il n'est pas possible de la généraliser à des tableaux contenant des éléments d'un autre type (n'oubliez pas que le tri à paniers ne fonctionne que sur des entiers). À titre d'exercice, vous pouvez recoder cette fonction dans votre langage préféré et tenter de la généraliser, afin de bien voir ce qui bloque.

Autrement, vous pouvez tester l'algorithme avec ce fichier d'exemple complet qui génère 10 entiers pseudo-aléatoirement et les trie.:

Code : C++

```
#include <algorithm>
#include <iostream>
#include <cstdlib>
#include <vector>

/* Le tri vu avant. Notez que l'argument vec est passé par
référence (&),
cela signifie que le tableau ne va pas être copié avant tri */
void tri_paniers(std::vector<int>& vec)
{
    int min = *std::min_element(vec.begin(),vec.end());
    int max = *std::max_element(vec.begin(),vec.end());

    std::vector<int> paniers(max-min+1);

    for(unsigned int i = 0; i < vec.size(); ++i)
        paniers[vec[i]-min]++;
```

```
        for(unsigned int i=0, c=0; i < paniers.size(); ++i)
            for(int j=0; j < paniers[i]; ++j, ++c)
                vec[c] = i+min;
    }

    void print_int(int n)
    {
        std::cout << n << std::endl;
    }

    int alea()
    {
        return rand()%10;
    }

    int main()
    {
        srand(time(NULL));

        /* On crée un vecteur de 10 entiers, et on le remplit du
    début (vec.begin()) à la fin (vec.end()) avec des nombres
    pseudo-aléatoires (tirés avec la fonction alea). */
        std::vector<int> vec(10);
        std::generate(vec.begin(),vec.end(),alea);

        /* On affiche chaque entier du vecteur, du début à la fin. */
        std::for_each(vec.begin(),vec.end(),print_int);

        std::cout << "Appuyez sur entrée pour débuter le tri...";
        getchar();

        /* On appelle le tri à paniers sur le vecteur, puis on le
    réaffiche, trié cette fois ci. */
        tri_paniers(vec);
        std::for_each(vec.begin(),vec.end(),print_int);

        return EXIT_SUCCESS;
    }
```

À titre d'exemple, voici une autre implémentation du tri à paniers, écrite cette fois-ci par bluestorm en OCaml, un des langages fonctionnels les plus connus :

Code : OCaml

```
    let tri_hist tab =
      let maxi, mini =
        let compare (m, m') e = max m e, min m' e in
        Array.fold_left compare (tab.(0), tab.(0)) tab in
      let paniers = Array.make (maxi - mini + 1) 0 in
      Array.iter (fun x -> paniers.(x - mini) <- paniers.(x - mini) + 1)
    tab;
      let k = ref 0 in
      let replace_paniers indice taille =
        Array.fill tab !k taille (indice + mini);
        k := !k + taille in
      Array.iteri replace_paniers paniers;
      tab
```

Petits calculs de complexité

Nous allons donc maintenant parler de la complexité de cet algorithme. Pour ceux qui ne connaissent pas, il s'agit de savoir combien d'opérations on doit faire pour trier un tableau de N chiffres. La complexité sert à évaluer la vitesse d'exécution d'un algorithme sur telle machine, mais aussi - et surtout - à comparer les algorithmes entre eux.

Le problème est qu'évaluer très précisément le nombre d'opérations effectuées est long et difficile. On va donc se contenter de calculer un ordre de grandeur, une variation, qui nous dira : « si vous doublez la longueur du tableau à trier, le temps d'exécution

quadruplera en moyenne » (par exemple).

Cette approche nous évite en outre d'avoir un nombre qui dépend de la machine sur laquelle on exécute le programme. En effet, si vous avez un ordinateur deux fois plus rapide que le mien, l'algorithme mettra deux fois moins de temps à s'exécuter chez vous : les mesures de temps que je pourrais vous donner ne vous serviraient donc à rien. Par contre, si je vous dit que le temps d'exécution croît comme le nombre d'entiers à trier, vous pourrez estimer le temps qu'il vous faudra pour trier vos chaussures par pointure.

Pour estimer cet ordre de grandeur, on néglige les instructions, très rapides à exécuter sur les machines récentes. On donc va surtout s'intéresser aux boucles, car ce sont elles qui monopolisent la majorité du temps d'exécution. Et, pour faire simple, on va définir qu'un tour de boucle vaut **1** opération. C'est la référence. Comme les boucles contiennent des instructions classiques qui sont super-rapides, il n'y a pas beaucoup de différence de l'une à l'autre (au niveau du temps), sauf dans le cas d'une boucle dans une boucle. Mais c'est encore une autre histoire  .

Revenons maintenant au tri à paniers. Imaginons que nous lui donnions un tableau de 20 chiffres. Que va-t-il se passer ?

- Il va déterminer le minimum et le maximum possibles dans le tableau. Il va donc parcourir une fois tout le tableau : il va faire 20 premières itérations.
- Il va allouer et initialiser un tableau de `(max-min)+1` éléments. On a donc une boucle de `(max-min)+1` tours qui se fait.
- Il va ensuite parcourir le tableau pour "remplir les paniers". Il parcourt alors le tableau du début à la fin : encore 20 itérations.
- Enfin, il va parcourir tous les paniers pour "remplir" le tableau. Comme il y a autant d'éléments au début et à la fin du tri, il y a donc, pour cette étape, autant d'itérations que de cases dans le tableau : 20.

Ainsi, on a `3*20 + (max-min)+1` tours de boucle, soit `60+tailleIntervalle`.
Dans le cas général, on remplace 20 par n'importe quel longueur N, et on dit que pour trier un tableau de N chiffres, il faut `3*N + (max-min)+1` tours de boucle. C'est une première estimation.

Cependant, ceci est encore trop « précis ». En effet, les complexités servent surtout à évaluer le comportement des algorithmes quand N devient très, très grand (plusieurs millions). À ce moment, le « fois 3 » devient ridicule devant N : on peut donc le négliger, de même que le « plus 1 » à la fin. Sur une machine moderne très rapide, on ne sentira pas la différence.

Petite parenthèse : si vous voulez vous amuser à calculer la complexité d'un algorithme récursif, la procédure que j'ai décrite ici ne fonctionne plus, vu que la notion de « boucle » n'a plus vraiment de sens. Il faut alors compter 1 récursion = 1 opération. Fin de la petite parenthèse.

Donc, quand on a `3*N` tours de boucles, on "arrondit" tout ça à N, tout court. Ce qui nous donne une complexité de `N + (max-min)`. En effet, on ne néglige pas le `max-min`, car, comme N, il peut devenir très, très grand dans certains cas.

On note donc que cet algorithme a une complexité `O(N + (max-min))`. Ce qui fait que la complexité augmente avec le nombre d'éléments et la disparité de la liste.

C'est relativement peu efficace par rapport aux autres algorithmes, notamment sur des listes tres disparates. Mais face à une liste d'éléments très proches les uns des autres (par exemple, les âges d'une classe), il peut servir. Cet algorithme peut même parfois se révéler bien plus rapide que les tris « classiques ».

Il faut aussi bien voir qu'un algorithme de tri dépendant de l'amplitude (différence entre le plus grand et le plus petit élément) de la liste considérée est assez original : la plupart ne dépendent que de la longueur de la liste. C'est un avantage ou un inconvénient selon les cas.
Voilà ! Vous savez maintenant trier des entiers avec cette nouvelle méthode, il ne vous reste plus qu'à l'implémenter  . Vous pouvez trouver plus d'informations sur l'article Wikipédia. Enfin, un ultime détail : il faut savoir que cet algorithme se nomme également tri par dénombrement, ou tri par comptage.

Pour toute question, remarque, jet de tomates, les commentaires sont ouverts.

Ce tutoriel a été corrigé par les zCorrecteurs.