Trier des tableaux en C++ : efficacité du **std::sort** (STL) et tris paramétrés

Hélène Toussaint, juillet 2014

Sommaire

1. Efficacité du std::sort

Le but de cette partie est de comparer l'algorithme de tri de la STL (**std::sort**) avec deux autres algorithmes de tri bien connus : le tri à bulle et le quick sort (itératif). Bien sûr comme l'efficacité des algorithmes de tri dépend en partie de la manière dont ils sont programmés l'implémentation du tri à bulle et du quick sort utilisés ici est donnée en annexe.

1.1. Conditions expérimentales

Les algorithmes sont testés sous Windows sur un PC avec un processeur Intel Core i7 3.4Ghz, 16 Go de RAM. Les programmes sont compilés sous VS 2010 en mode release. Les algorithmes ont également été testés sous linux (les résultats ne sont pas reportés ici, ils sont très similaires). Les tests sont effectués sur des tableaux d'entiers initialisés de manière aléatoire.

1.2. Tableaux de grande taille

Trois tableaux sont initialisés de manière identique, la table 1 donne le temps en ms pour effectuer le tri d'un tableau en utilisant :

- le tri à bulle (colonne 2)
- le quick sort (colonne 3)

- le **`std::sort`** de la STL (colonne 4)

taille du tableau à trier	cpu (ms) tri à bulle	cpu (ms) quick sort	cpu (ms) **`std::sort`** (STL)
1000	< 10 ms	< 10 ms	< 10 ms
2000	10	< 10 ms	< 10 ms
4000	10	< 10 ms	< 10 ms
8000	70	< 10 ms	< 10 ms
16000	350	< 10 ms	< 10 ms
32000	1490	< 10 ms	< 10 ms
64000	6100	10	< 10 ms
128000	24700	10	< 10 ms
256000	99280	30	20
512000	397760	100	20
1024000	-	330	40
2048000	-	1240	80
4096000	-	4770	150
8192000	-	18740	310
16384000	-	74389	620
32768000	-	295401	1240

Tableau 1 : comparaison des temps de calcul (en ms) sur des grandes tailles de tableau.

1.3. Tableaux de petite taille

On reprend le même procédé que précédemment mais on effectue chaque tri 10^7 fois (sinon les temps de calcul sont trop faibles pour être comparés !). Les tableaux sont réinitialisés avant chaque tri, la réinitialisation du tableau est prise en compte dans le temps de calcul mais elle est identique pour chaque méthode.

taille du tableau à trier	cpu (ms) tri à bulle	cpu (ms) quick sort	cpu (ms) **`std::sort`** (STL)
4 (10^7 tris)	90	720	120
8 (10^7 tris)	310	880	220
16 (10^7 tris)	1070	1500	770
32 (10^7 tris)	4240	2430	2490
64 (10^7 tris)	18950	5270	5790

Tableau 2 : comparaison des temps de calcul (en ms) sur des petites tailles de tableau (10^7 tris pour chaque tableau)

1.4. Tableaux presque triés

On teste le **`std::sort`** sur un tableau presque trié : à partir d'un tableau trié on change aléatoirement la valeur de 10 éléments et on effectue de nouveau le tri du tableau.

taille du tableau	1024000	2048000	4096000	8192000	16384000	32768000
cpu (ms)	< 10 ms	15	16	47	93	187

Tableau 3 : performance du std::sort sur un tableau "presque" trié

Remarque : sur les tableaux déjà complètement triés le **`std::sort`** fonctionne également très bien, il est aussi rapide que le tri à bulle qui n'effectue qu'un seul passage sur le tableau. Notre quick sort n'est pas du tout approprié aux tableaux déjà triés étant donné qu'on prend comme pivot le premier élément du tableau.

1.5. Synthèse

Il apparaît dans nos tests que le tri **`std::sort`** de la STL donne les meilleurs temps cpu sur presque toutes les tailles de tableau : il est beaucoup plus performant que notre quick sort sur les très grands tableaux (> 10^5 éléments) et sur les très petits (< 20 éléments). Sur les tableaux de taille moyenne les temps cpu des deux méthodes sont assez proches.

On remarque que le tri à bulle est meilleur que le quick sort sur les très petites tailles de tableau. Cependant sa complexité en $O(n^2)$ le rend inadapté au tri de tableau de moyenne et grande taille.

2. Tri d'un tableau en fonction d'un autre

On est parfois amené à trier un tableau en fonction d'un autre : par exemple ordonner un ensemble d'objets (numérotés) en fonction de leur poids stocké dans un tableau auxiliaire. Dans cette section on envisage différentes méthodes et structures de données pour trier ces objets (en utilisant les outils mis à notre disposition par la STL) et on compare leur efficacité en terme de temps de calcul.

On suppose ici que les objets sont identifiés par un entier i ($i \in \{1, \dots, n\}$) et que leur poids est stocké dans un tableau : **`poids[i]`** donne le poids de l'objet i.

2.1. Conditions expérimentales

Comme précédemment, les algorithmes sont testés sous Windows sur un PC avec un processeur Intel Core i7 3.4Ghz, 16 Go de RAM. Cependant pour cette partie les programmes sont compilés sous VS 2012 (pour pouvoir utiliser les *lambda expressions* - C++11) en mode release.

Pour chaque Structure De Données (SDD) testée le temps de calcul prend en compte :
1. l'initialisation de la SDD
2. le tri de la SDD
3. l'utilisation de la SDD (somme des poids des n / 2 meilleurs objets)

L'identifiant (ou numéro) d'un objet est un entier, son poids est un réel.

2.2. Présentation des *lambda expressions* (C++11)

On présente ici rapidement (et de manière non exhaustive) les *lambda expressions* (nouvelle fonctionnalité offerte par C++11) qui vont être utilisées dans la section suivante.

Une *lambda expression* peut être vue comme une mini fonction que l'on peut définir directement à l'endroit où on en a besoin sans lui donner de nom. De plus elle connaît implicitement les variables qui sont définies au moment où on l'utilise (sans qu'il soit nécessaire de lui passer en paramètre).

On la définie de la manière suivante :

```
[ type d'accès aux variables externes ] ( paramètres ) { code }
```
Prototype d'une lambda expression

Une lambda expression peut accéder aux variables externes soit par référence (elle commencera alors par **[&]**) soit par copie (elle commencera alors par **[=]**). Si elle n'utilise pas de variable externe il suffit de laisser les crochets vides (**[]**). On peut aussi préciser le type d'accès pour chaque variable externe (ex : **[&var1, var2]**).

Les paramètres et le code sont définis comme pour une fonction classique.

```
sort(job.begin(), job.end(), [&poids](const int i, const int j)
                                { return (poids[i] < poids[j]); } );
```
Exemple d'utilisation d'une lambda expression comme opérateur de comparaison pour le **std::sort.** Le vecteur **poids** est défini dans la fonction qui appelle **std::sort**, il est passé par référence à la lambda expression.

Remarque : dans l'exemple précédent il n'aurait pas été possible d'utiliser un opérateur de comparaison classique car on a besoin du vecteur **poids** qui ne peut pas être passé en paramètre (un opérateur de comparaison accepte seulement 2 paramètres : les 2 objets à comparer).

2.3. Programmes et SDD utilisés

On suppose disposer d'un vecteur de *doubles* **poids** initialisé correctement.

```
        vector< pair<double,int> > vp;
        //init
        somme1 = 0;
        for (int i = 0; i < N; ++i)
            vp.push_back( pair<double,int> (poids[i], i) );
        //tri
        sort(vp.begin(), vp.end());
        //utilisation : somme des poids des N/2 plus petits
        for (int i = 0; i < Nsur2; ++i)
            somme1 += poids[vp[i].second];
```
Exemple 1 : utilisation d'un **vector< pair < double, int > >**

```
        set < pair<double,int> >sp;
        //init et tri (insertion triee avec le set)
        somme3 = 0;
        for (int i = 0; i < N; ++i)
            sp.insert( pair<double,int> (poids[i], i) );
        //utilisation : somme des poids des N/2 plus petits,
        set< pair<double,int> >::iterator it = sp.begin();
        for (int i = 0; i < Nsur2; ++i)
        {
            somme3 += poids[it->second];
            it++;
        }
```
Exemple 2 : utilisation d'un **set< pair < double, int > >**

```
vector< int > job ;
//init
somme2 = 0;
for (int i = 0; i < N; ++i)
    job.push_back(i);
//tri
sort(job.begin(), job.end(), [&poids](const int i, const int j)
                                        { return (poids[i] < poids[j]); });
//utilisation : somme des poids des N/2 plus petits
for (int i = 0; i < Nsur2; ++i)
    somme2 += poids[job[i]];
```

Exemple 3 : utilisation d'un **vector< int >** et std::sort paramétré par une *lambda expression* (voir section 2.3)

```
class ObjPondere
{
public:
    int _numObj;
    double _poids;

    ObjPondere (int i, double pi ) : _numObj(i), _poids(pi){}

    bool operator < ( const  ObjPondere & obj2)
    {
        return _poids < obj2._poids;
    }
};

//----------utilisation de la classe-------------------

vector< ObjPondere > vobj;
somme4 = 0;
for (int i = 0; i < N; ++i)
    vobj.push_back( ObjPondere(i, poids[i]) );
//tri
sort( vobj.begin(), vobj.end()

//utilisation : somme des poids des N/2 plus petits,
for (int i = 0; i < Nsur2; ++i)
    somme4 += vobj[i]._poids;
```

Exemple 4 : utilisation d'une classe **ObjPondere** redéfinissant l'opérateur de comparaison

2.4. Comparaison des temps de calcul

	cpu (ms) méthode 1 **vector < pair >**	cpu (ms) méthode 2 **set < pair >**	cpu (ms) méthode 3 lambda exp.	cpu (ms) méthode 4 classe et redéf. op. <
N = 20 (10^6 tests)	296	1716	**234**	250
N = 50 (10^6 tests)	733	4930	**639**	686
N = 100 (10^6 tests)	2043	10140	**1700**	1545
N = 10^6 (1 test)	140	672	134	**94**
N = 10^7 (1 test)	1576	13369	1934	**936**

2.5. Synthèse

On constate que le **std::set** n'est pas du tout performant pour trier des objets (en plus il supprime les doublons, attention !). On retient donc uniquement les solutions utilisant un **std::vector** trié par **std::sort**. Le plus efficace est soit de définir une classe contenant les infos dont on a besoin (et en plus

c'est très pratique pour encapsuler les données !) en redéfinissant l'opérateur " > " pour cette classe, soit d'utiliser une lambda expression comme opérateur de comparaison dans le **std::sort** (ce qui requiert un compilateur pour le C++11 : VS 2012 minimum pour Windows ou gcc 4.7 minimum).

3. Annexe : algorithmes de tri utilisés pour la partie 1

```
void quickSort(int * tab, int n) //algo principal quick sort
{
	int deb, fin, j;
	bool fini = false;

	//================================================
	//creation pile
	int * pile = new int [2*n];
	int sommetPile = -1; deb = 0; fin = n-1;

	//============================================================
	// boucle principale
	while ( !fini )
	{
		while ( deb < fin )
		{
			j = place(tab, deb, fin);
			if (deb < j - 1)
			{
				//empile partie gauche
				pile[sommetPile+1] = deb;
				pile[sommetPile+2] = j-1;
				sommetPile = sommetPile + 2;
			}
			//traite partie droite
			deb = j+1;
		}
		if (sommetPile > 0) //pile non vide, on depile
		{
			fin = pile[sommetPile];
			deb = pile[sommetPile-1];
			sommetPile = sommetPile-2;
		}
		else
			fini = true;
	}
	delete [] pile;
}
//sous-procédure place :
int place(int * tab, int debut, int fin)
{
	int pivot = tab[debut];
	int gauche = debut+1;
	int droite = fin;
	int tmp;

	//on met a droite les elements sup au pivot et a gauche les elem inf :
	while( gauche <= droite )
	{
		while(  gauche <= droite && (tab[gauche] <= pivot) )
			gauche++;
		while(  gauche <= droite && (tab[droite] > pivot) )
			droite--;
		if( gauche < droite )
		{
			tmp = tab[droite];
			tab[droite] = tab[gauche];
			tab[gauche] = tmp;
			gauche++;
			droite--;
		}
	}

	//mettre le pivot a sa place
	tab[debut] = tab[droite];
	tab[droite] = pivot;

	return droite;
}
```

Algo 1 : Quick sort

```
void triBulle(int * tab, int n)
{
	bool stop;
	int tmp, j = 0;

	do
	{
		stop = true;
		//parcours le tab. de la fin vers le debut, au jeme passage les j premiers elements
		  sont tries
		for( int i = n-1; i > j; --i )
		{
			if( tab[i] < tab[i-1]) //l element le plus petit remonte vers le debut
			{
				tmp = tab[i];
				tab[i] = tab[i-1];
				tab[i-1] = tmp;
				stop = false;
			}
		}
		j++;
	}
	while(!stop);//si pas de changement à un passage -> tab trie
}
```

Algo 2 : Tri à bulle