Tris en C et C++

Tous les langages de programmation fournissent des algorithmes de tri standard. Il est important d'en comprendre les propriétés.

Langage C

En C, la librairie "`stdlib.h`" fournit la fonction `qsort` (http://www.cplusplus.com/reference/cstdlib/qsort/), dont le prototype est

In [1]:
```
void qsort (void* base,
            size_t num,
            size_t size,
            int (*comp)(const void*,const void*));
```

- `void *base` : adresse du premier élément du tableau
- `size_t num` : nombre d'éléments à trier

Ecrivons une fonction comparant deux entiers, de prototype

In [2]:
```
int plus_petit (const void * a, const void * b);
```

Elle doit

- caster les pointeurs `void*` en pointeurs `int*`
- comparer les deux valeurs entières
- retourner un entier
  - `<0` si `*a` est plus petit que `*b`
  - `>0` si `*b` est plus petit que `*a`
  - `0` s'ils sont égaux selon le critère choisi

In [3]:
```
int plus_petit (const void * a, const void * b)
{
  return ( *(const int*)a - *(const int*)b );
}
```

Soit le tableau d'entiers

In [4]:
```
int values[] = { 40, 10, 100, 90, 20, 25 };
int N = sizeof(values)/sizeof(int);
```

Pour le trier entièrement, il suffit d'écrire

In [5]:
```
#include "stdlib.h"
qsort (values, N, sizeof(int), plus_petit);
```

Ce qui donne le tableau trié suivant

In [6]:
```
values
```

Out[6]:
```
{ 10, 20, 25, 40, 90, 100 }
```

Propriétés

Le langage C ne fournit pas de garantie sur la complexité de `qsort`, mais en pratique

- `qsort` est le diminutif de Quick Sort, l'algorithme de tri rapide
- Le tri a donc une complexité moyenne $\Theta(n \log n)$
- Il peut avoir une complexité quadratique $\Theta(n^2)$ dans le pire des cas
- Il n'est pas stable

Tri rapide en C++

En C++, la librairie <algorithm> fournit la fonction std::sort (http://www.cplusplus.com/reference/algorithm/sort/), dont les prototypes sont

In [7]:
```
template <class RandomAccessIterator>
void sort (RandomAccessIterator first,
           RandomAccessIterator last);
```

- first : itérateur vers le premier élément à trier
- last : itérateur vers l'élément qui suit le dernier à trier. Ensemble, ils définissent une séquence [first,last[.

In [8]:
```
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first,
           RandomAccessIterator last,
           Compare comp);
```

- first : itérateur vers le premier élément à trier
- last : itérateur vers l'élément qui suit le dernier à trier. Ensemble, ils définissent une séquence [first,last[.
- comp : fonction de comparaison. Prend deux éléments en paramètres et retourne un bool qui vaut vrai si le premier est plus petit que le second. Si elle n'est pas spécifiée, le tri utilise l'opérateur < du type trié

Soit le vecteur v dont on veut trier les 4 premiers éléments

In [9]:
```
#include <vector>
std::vector<int> v{ 32,71,12,45,26,80,53,33 };
```

In [10]:
```
#include <algorithm>
std::sort( v.begin(), v.begin()  + 4 );
v
```

Out[10]:
```
{ 12, 32, 45, 71, 26, 80, 53, 33 }
```

Pour trier tout le vecteur, on écrit

In [11]:
```
std::sort( v.begin(), v.end() );
v
```

Out[11]:
```
{ 12, 26, 32, 33, 45, 53, 71, 80 }
```

La version avec comparaison générique peut prendre en paramètre une fonction,

In [12]:
```
bool plus_grand (int i,int j) { return (i>j); }
```

In [13]:
```
std::sort(v.begin(), v.end(), plus_grand);
v
```

Out[13]:
```
{ 80, 71, 53, 45, 33, 32, 26, 12 }
```

un objet fonction, ou foncteur (https://en.wikipedia.org/wiki/Function_object), c'est à dire une structure ou une classe qui définit l'opérateur `operator()`,

In [14]:
```
struct plus_petit_modulo {
    int base;
    bool operator() (int i,int j) { return (i%base < j%base );}
};
```

In [15]:
```
auto compare_dernier_chiffre = plus_petit_modulo{10};
std::sort( v.begin(), v.end(), compare_dernier_chiffre );
v
```

Out[15]: { 80, 71, 32, 12, 53, 33, 45, 26 }

In [16]:
```
auto compare_parite = plus_petit_modulo{2};
std::sort( v.begin(), v.end(), compare_parite );
v
```

Out[16]: { 80, 32, 12, 26, 71, 53, 33, 45 }

ou une expression lambda (https://en.wikipedia.org/wiki/Anonymous_function) (C++11), i.e. un objet fonction anonyme capable de capturer des variables dans la portée.

In [17]:
```
int B = 2;
std::sort( v.begin(), v.end(),
          [&B](int i, int j) { return i%B < j%B; } );
v
```

Out[17]: { 80, 32, 12, 26, 71, 53, 33, 45 }

Propriétés

La librairie standard garantit une **complexité moyenne** linéarithmique $\Theta(n \log n)$.

En pratique, `std::sort` est toujours une variation de l'algorithme de **tri rapide**, éventuellement avec un tri par insertion pour les partitions les plus petites.

Donc,

- il n'est pas stable
- c'est le plus rapide des tris proposés par la STL
- on ne peut exclure une complexité quadratique dans le pire des cas

Le tri rapide étant un tri par échange, il est important que la fonction `swap(T,T)` soit efficace pour le type `T` à trier.

Tri stable en C++

La librairie `<algorithm>` fournit également la fonction `std::stable_sort` (http://www.cplusplus.com/reference/algorithm/stable_sort/), dont les prototypes sont

In [18]:
```
template <class RandomAccessIterator>
void stable_sort (RandomAccessIterator first,
                  RandomAccessIterator last );
```

- `first` : itérateur vers le premier élément à trier
- `last` : itérateur vers l'élément qui suit le dernier à trier.
  Ensemble, ils définissent une séquence `[first,last[`.

In [19]:
```
template <class RandomAccessIterator, class Compare>
void stable_sort (RandomAccessIterator first,
                  RandomAccessIterator last,
                  Compare comp );
```

- `first` : itérateur vers le premier élément à trier
- `last` : itérateur vers l'élément qui suit le dernier à trier. Ensemble, ils définissent une séquence `[first,last[`.
- `comp` : fonction de comparaison. Prend deux éléments en paramètres et retourne un `bool` qui vaut vrai si le premier est plus petit que le second.

Trions par exemple en ne tenant compte que des parties entières avec la fonction

In [20]:
```
bool partie_entiere_plus_petite (double i,double j)
{ return (int(i) < int(j)); }
```

In [21]:
```
std::vector<double> v2 {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.6
2, 2.58};
std::stable_sort(v2.begin(), v2.end(), partie_entiere_plus_peti
te);
v2
```

Out[21]:
```
{ 1.41, 1.73, 1.32, 1.62, 2.72, 2.58, 3.14, 4.67 }
```

On voit que pour une partie entiére donnée, l'ordre des éléments originaux est conservé.

Mais attention, il est indispensable que la fonction de tri mette en oeuvre une inégalité stricte. Sinon la stabilité n'est pas garantie.

In [22]:
```
bool partie_entiere_plus_petite_ou_egale (double i,double j)
{ return (int(i) <= int(j)); }
```

In [23]:
```
std::vector<double> v3 {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.6
2, 2.58};
std::stable_sort(v3.begin(), v3.end(), partie_entiere_plus_peti
te_ou_egale);
v3
```

Out[23]:
```
{ 1.62, 1.32, 1.73, 1.41, 2.58, 2.72, 3.14, 4.67 }
```

Propriétés

La librairie standard garantit

- une complexité temporelle linéarithmique $\Theta(n \log n)$, si il y a assez de mémoire.
- une complexité temporelle $\Theta(n \log^2 n)$ si le tri fusion doit être réalisé en place.

En pratique, `std::stable_sort` est toujours une variation de l'algorithme de **tri par fusion**.

Donc,

- il est stable
- il est moins rapide `std::sort`
- les complexités sont également garanties dans le pire des cas

Le tri par fusion effectant de nombreux déplacements, il est important que l'affectation `T = std::move(T)` soit efficace pour le type `T` à trier.

n^{ieme} élément en C++

La librairie <`algorithm`> fournit aussi une fonction de sélection rapide sous le nom de `nth_element` (http://www.cplusplus.com/reference/algorithm/nth_element/), dont le prototype est

In [24]:
```
template <class RandomAccessIterator>
void nth_element (RandomAccessIterator first,
                  RandomAccessIterator nth,
                  RandomAccessIterator last);
```

où les éléments à traiter sont dans l'intervalle `[first,last[` et la valeur de `n` est donnée par `n = nth - first`.

La fonction met le n^{ieme} élément à sa place,

mais a aussi pour effet de partitionner `[first,last[` autour de sa valeur.

- `[first,nth[` ne contient que des éléments ≤ `*nth`
- `[nth,last[` ne contient que des éléments ≥ `*nth`

Il existe évidemment aussi une version avec fonction de comparaison générique.

In [25]:
```
template <class RandomAccessIterator, class Compare>
void nth_element (RandomAccessIterator first,
                  RandomAccessIterator nth,
                  RandomAccessIterator last,
                  Compare comp);
```

Trouvons le 4^{ieme} élément (d'indice 3) du tableau v4 et partitionnons le autour de cette valeur

In [26]:
```
std::vector<int> v4{ 3,5,2,6,8,1,7,4};
std::nth_element(v4.begin(), v4.begin()+3, v4.end());
v4
```

Out[26]:
```
{ 3, 1, 2, 4, 6, 5, 7, 8 }
```

Propriétés

La STL garantit une complexité **moyenne** linéraire $\Theta(n)$ pour `n = last - first`.

Cela correspond à la complexité de l'algorithme de sélection rapide.

Tri partiel en C++

Il est également possible de ne trier qu'une partie d'un tableau avec la fonction `partial_sort` (http://www.cplusplus.com/reference/algorithm/partial_sort/) de la librairie `<algorithm>`. Le prototype en est

In [27]:
```
template <class RandomAccessIterator>
void partial_sort (RandomAccessIterator first,
                   RandomAccessIterator middle,
                   RandomAccessIterator last );
```

- `[first,last[` est l'intervalle des valeurs à trier.
- `middle` doit être dans cet intervalle.

En sortie, l'intervalle `[first,middle[`

- contient les `middle-first` plus petites valeurs
- est trié

l'intervalle `[middle,last[`

- ne contient que des valeurs plus grandes
- dans un ordre quelconque.

La fonction de tri peut également être générique

In [28]:
```
template <class RandomAccessIterator, class Compare>
void partial_sort (RandomAccessIterator first,
                   RandomAccessIterator middle,
                   RandomAccessIterator last,
                   Compare comp );
```

Trions les 4 plus petits éléments du tableau v5.

In [29]:
```
std::vector<int> v5{ 4,3,6,2,7,1,8,5};
std::partial_sort(v5.begin(), v5.begin()+4, v5.end());
v5
```

Out[29]:
```
{ 1, 2, 3, 4, 7, 6, 8, 5 }
```

Trions les 3 éléments du tableau v6 de plus grande valeur absolue.

In [30]:
```
bool plus_grande_valeur_absolue(double a, double b) {
    return abs(a) > abs(b);
}
```

In [31]:
```
std::vector<double> v6{ 4, -3, -6, 2, -7, -1, 8, 5};
std::partial_sort(v6.begin(), v6.begin()+3, v6.end(), plus_gran
de_valeur_absolue);
v6
```

Out[31]:
```
{ 8, -7, -6, 2, -3, -1, 4, 5 }
```

Propriétés

La librairie standard garantit une complexité $\Theta(n \log m)$ avec

- `n = last - first`
- `m = middle - first`.

Il existe plusieurs possibilités pour mettre en oeuvre ce tri partiel.

- La complexité garantie suggère
  - l'utilisation d'un tas de m éléments
  - dans lequel on insère les autres $n - m$ éléments
  - en supprimant le maximum entre chaque insertion.

- Une approche alternative consiste à
  - utiliser la sélection rapide pour trouver le m^{ieme} élément
  - ce qui paritionne aussi le tableau de sorte que les m plus petits sont à gauche
  - utiliser le tri rapide sur ces m plus petits éléments

Cette approche alternative a une complexité $\Theta(n + m \log m)$ en moyenne, ce qui est meilleur en moyenne, mais éventuellement moins bon dans le pire des cas.

Et encore ...

Notons enfin l'existence d'autres fonctions mettant en oeuvre une partie des tris efficaces (partition et fusion) et testant l'état trié / partitionné d'une séquence d'éléments. Leurs noms sont explicites

heig-vd

ASD1 Notebooks on GitHub.io (https://ocuisenaire.github.io/ASD1-notebooks/)