

Librairies

Stable_sort (d,f,[comp])

Trie selon la comparaison (ordre croissant si rien), si valeurs égales ou pas sur du tri, on garde l'ordre de gauche à droite.

sort (d,f,[comp])

trie selon la comparaison (ordre croissant si rien), mais ne garanti pas l'ordre des valeurs en cas de confusion. Parfois indéterminé. (Souvent si utilisé APRES un stable_sort()).

Qsort

```
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
- size_t size : taille d'un élément du tableau
- int (*comp) (void const *a, void const *b) : adresse de la fonction de comparaison, fournie par l'utilisateur.

Complexité moyenne $O(n \log n)$, pire cas $O(n^2)$

Instable

```
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

Sort

$O(n \log n)$ en moyenne, voir même quadratique dans le pire des cas

Instable

Std::swap(T,T) doit être efficace !

C'est une variation du tri par fusion.

Stable_sort

$O(n \log n)$ si pas assez de mémoire, $O(n \log^2 n)$ si le tri fusion doit être réalisé en place.

Stable

Std::move(T) doit être efficace !

C'est une variation de la sélection rapide

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

```
{ 3, 1, 2, 4, 6, 5, 7, 8 }
```

< 4 > 4

nth element

Complexité moyenne $O(N)$

Partial_sort(p,middle,d)

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

```
{ 1, 2, 3, 4, 7, 6, 8, 5 }
```

Trié Modifié

begin()+4

$O(n \log m)$ ou $O(n+m \log m)$

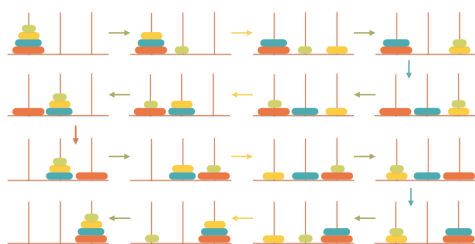
Complexités

Ordre de croissance : $\log(n) < n < n^2 < n^3 < 2^n$

Nom	Complexité
Arbre plein	$O(\log(n))$
rand % N	$1/N$
rand % 2	$1/2$
rand % N == 0	$O(1)$
i*i	\sqrt{n}
i*=2 ou i/= 2	$O(\log N)$
M or N	$O(\max(N,M))$
M and N	$O(\min(N,M))$
i = N; i< M	$O(\max(0, M-N))$
f(N+1)	$O(n)$
f(N-1) + f (N-1)	$O(2^N)$
f(N/2) = f(N*2)	$O(\log N)$
f(N-1) + f (N-2)	ϕ^N , $\phi = (\sqrt{5} + 1) / 2$
Euclide(a,b) : (b, a%b)	$O(\log(\min(a,b)))$


Algorithmes

Accumulate()	$O(N)$
Sort()	$O(\log N)$
Std::max_element()/	$O(N)$
Std::min_element()	
Std::nth_element	$O(N)$ ou pire : $O(N^2)$
Std::lower_bound() /	$O(\log(N))$
Std::upper_bound()	
Std::equal_range()	$O(2 * \log(N))$
Std::generate()	$O(N)$
Std::sort()	$O(N)$
Std::stable_sort()	
Set : insertion	$O(\log N)$
: find	$O(1)$



Stable	<p>Ne swap pas deux valeurs identiques, préserve l'ordre initial des éléments égaux.</p> <p>On fait de gauche à droite un swap si $val > val_{suivante}$, on ne revient pas en arrière. On recommence une fois arrivé à la fin.</p>
Bulle	<p>$O(n^2)$ comparaison / $O(0 \text{ à } n^2)$ échanges selon l'ordre des éléments à trier</p> <p>Stable, Le tri à bulle n'est stable que s'il n'échange pas les éléments égaux</p> <p>On part depuis la gauche et on swap la valeur actuelle avec la plus petite présente à la suite. Pré-trié on voit les valeurs de gauche dans le bon ordre.</p>
Sélection	<p>$O(n^2)$ comparaison / $O(n)$ échanges, quelle que soit l'ordre d'entrée</p> <p>Instable car il peut swap deux même valeur (1 dans la zone triée, 1 dans la zone non triée) T</p> <p>Depuis la gauche, on regarde si $val > val_{suivante}$, si oui, on réécrit la valeur plus grande 2 fois, une sous elle-même et une sous la valeur à swap, ensuite on vérifie si la valeur swap est plus petite que son nouveau voisin, si oui on réécrit 2 fois la valeur du voisin plus grand et on le fait jusqu'à ce que le voisin gauche soit plus petit ou inexistant. Finalement on écrit la ligne triée puis on recommence.</p> <p>Pré-trié il est quasi trié à gauche et intouché à droite.</p> <p>$O(n)$ si déjà trié, $O(n^2)$ Trié à l'envers - $O(n^2)$ Entrée aléatoire</p> <p>Stable</p> <p>On couple le tableau en 2, et on recoupe en 2 si possible, ensuite on trie un groupe à la fois de gauche à droite, puis arrivé à la moitié on écrit la ligne triée, on fait de même avec la droite puis on fusionne les 2 parties ensemble.</p> <p>Pré-trié la moitié gauche et droite sont triées.</p>
Fusion	<p>$O(n \log(n))$</p> <p>Stable</p> <p>On prend le pivot et on le swap avec la dernière valeur, puis depuis la gauche, on vérifie depuis la gauche que $val(i) < pivot$ et depuis la droite que $val(j) > pivot$, si non on swap les valeurs. Si $A(i)$ et $A(j)$ se croisent on swap la valeur du croisement avec le pivot.</p> <p>Partition on fait $< ou >$</p> <p>Tri on fait $<= ou >=$</p> <p>$O(n \log(n))$ moyen, $O(n^2)$ pire des cas</p> <p>Stable si lors de la partition : $<= ou >=$</p> <p>Instable si lors de la partition : $< ou >$ (Instable généralement utilisé)</p>
Partition rapide	
Sélection Rapide	<p>$O(N \log N)$ en triant le tableau, $O(N)$ pour les cas simples ($k=1, k=N, \dots$),</p>
Tri comptage	<p>Complexité $O(n+b)$ pour n éléments pouvant prendre b valeurs distinctes</p> <p>Stable</p> <p>Un tableau pour les compteurs</p> <p>Un tableau pour la sortie</p>
Tri par base	<p>Complexité $O(d \cdot (n+b))$ pour n éléments pouvant prendre bd valeurs</p> <p>2.d passages sur le tableau : $O(d \cdot n)$</p> <p>d passages sur les compteurs : $O(d \cdot b)$</p> <p>Stable</p>

Factorielle récursif	$O(n)$
Factorielle itératif	$O(n)$
Fibonacci récursif	$O(1.618^n)$
Fibonacci itératif	$O(n)$
PGCD (Euclide)	$O(\log(n))$
Tours de Hanoi récursif	$O(2^n)$
Tours de Hanoi itératif	$O(2^n)$
Permutations	$O(n!)$
Tic Tac Toe	9!
Puissance 4, profondeur d'exploration de d tours	$O(7^d)$
Minimax (negamax), m mouvements possibles par tour, profondeur de d tours	$O(m^d)$

<pre> fonction BubbleSort(A,n) (tableau A de n éléments) pour i de 1 à n-1 boucler pour j de 1 à n-i boucler si A(j+1) < A(j), alors permuter A(j) et A(j+1) fin si fin pour j fin pour i </pre>	<pre> fonction SelectionSort(A,n) pour i de 1 à n-1 boucler imin ← i pour j de i+1 à n boucler si A(j) < A(imin), alors imin ← j fin si fin pour j permuter A(i) et A(imin) fin pour i </pre>	<pre> fonction InsertionSort(A,n) pour i de 2 à n boucler tmp ← A(i) j ← i tant que j-1 ≥ 1 et A(j-1) > tmp boucler A(j) ← A(j-1) décrémenter j de 1 fin tant que A(j) ← tmp fin pour i </pre> 
<pre> fonction TriFusion(A,lo,hi) si hi <= lo, alors retourner fin si mid ← lo + (hi-lo)/2 TriFusion(A,lo,mid) TriFusion(A,mid+1,hi) Fusionner(A,lo,mid,hi) </pre>	<pre> fonction Partition(A,lo,hi) i ← lo-1, j ← hi boucler répéter incrémenter i tant que A(i) < A(hi) répéter décrémenter j tant que j>lo et A(hi) < A(j) si i ≥ j, alors sortir boucle fin si permuter A(i) et A(j) fin boucler permuter A(i) et A(hi) retourner i </pre>	<pre> fonction TriRapide(A,lo,hi) si lo < hi, alors p ← choisir l'élément pivot permuter A(hi) et A(p) i ← Partition(A,lo,hi) TriRapide(A,lo,i-1) TriRapide(A,i+1,hi) fin si </pre>
<pre> fonction SelectionRapide(A,n,k) lo ← 1 hi ← n tant que hi > lo i ← partition(A,lo,hi) si i < k, alors lo ← i+1 sinon, si i > k, alors hi ← i-1 sinon (i = k) retourner A(k) fin tant que retourner A(k) </pre>	<pre> fonction TriComptage(A,n,b,clé): C ← tableau de b compteurs à zéro pour tout e dans A C[clé(e)] += 1 idx ← 1 pour i de 1 à b tmp ← C[i] C[i] ← idx idx += tmp B = tableau de même taille que A pour tout e dans A B[C[clé(e)]] ← déplacer e C[clé(e)] += 1 return B </pre>	<pre> fonction triParBase(T, d): Pour i allant de d à 1 Trier avec un tri stable le tableau T selon le i-ème chiffre </pre>

<p>Comparaison pour qsort()</p> <pre> int plus_petit (const void * a, const void * b) { return (*(const int*)a - *(const int*)b); } </pre> <p>Elle doit</p> <ul style="list-style-type: none"> • caster les pointeurs <code>void*</code> en pointeurs <code>int*</code> • comparer les deux valeurs entières • retourner un entier <ul style="list-style-type: none"> ▪ <0 si <code>*a</code> est plus petit que <code>*b</code> ▪ >0 si <code>*b</code> est plus petit que <code>*a</code> ▪ 0 s'ils sont égaux selon le critère choisi 		<p>Utilisation qsort()</p> <pre> int values[] = { 40, 10, 100, 90, 20, 25 }; int N = sizeof(values)/sizeof(int); </pre> <p>Pour le trier entièrement, il suffit d'écrire</p> <pre> #include "stdlib.h" qsort (values, N, sizeof(int), plus_petit); </pre> <pre> template <typename T> void swap(T& lhs, T& rhs) { T tmp = lhs; // constructeur de copie lhs = rhs; // operateur d'affectation rhs = tmp; // operateur d'affectation } // destruction automatique de tmp </pre>	
<pre> template <typename Iterator> void BubbleSort(Iterator first, Iterator last) { if(first == last) return; size_t N = distance(first,last); for(size_t i = 1; i != N; ++i) for (auto j = first; j != prev(last-i); ++j) if (*next(j) < *j) swap(*j, *next(j)); } </pre>	<p>Méthode</p> <pre> void swap(RandomString& other) noexcept { using std::swap; swap(this->N, other.N); swap(this->data, other.data); } </pre> <p>Fonction</p> <pre> void swap(RandomString& lhs, RandomString& rhs) noexcept { lhs.swap(rhs); } </pre>	<pre> template <typename Iterator> // bidirectionnal iterator void InsertionSort(Iterator first, Iterator last) { if (first == last) return; for (Iterator i = next(first); i != last; ++i) { auto tmp = std::move(*i); // constructeur de déplacement Iterator j = i; while (j != first and tmp < *prev(j)) { *j = std::move(*prev(j)); // affectation par déplacement --j; } *j = std::move(tmp); // affectation par déplacement } } </pre>	