

École Polytechnique

INF411

Les bases de la programmation et de l'algorithmique

Jean-Christophe Filliâtre

Bloc 8 / lundi 22 octobre 2018

quels sont les dix mots les plus fréquents dans
Le tour du monde en quatre-vingts jours ?

on peut le faire avec le *shell*, en une ligne !

```
cat file | tr -cs '[:alpha:]' '\n*' | sort | uniq -c | sort -rn
```

```
2826 de
1616 le
1587 la
1472 et
1112 il
 952 les
 832 un
 788 en
 766 du
 670 Fogg
...
```

on envoie le fichier sur la sortie standard

```
cat ltdme80j.txt
```

on remplace les caractères non-alphabétiques par un retour-charriot

```
| tr -cs '[:alpha:]' '\n*'
```

on trie les lignes par ordre alphabétique

```
| sort
```

on regroupe les lignes identiques, en les comptant (-c)

```
| uniq -c
```

on trie numériquement (-n), en ordre inverse (-r)

```
| sort -rn
```

on a utilisé ici deux tris

- un tri par ordre alphabétique (de 76 420 lignes)
- un tri numérique (de 8 608 lignes)

l'objet de l'amphi d'aujourd'hui est justement **le tri**

mélanger

comment mélanger N valeurs correctement ?

par exemple les N éléments d'un tableau

rappel : le mélange de Knuth (*Knuth shuffle*)

chaque élément i est échangé avec un élément $j \in [0, i]$

```
static<E> void shuffle(E[] a) {
    for (int i = 1; i < a.length; i++) {
        int j = (int)(Math.random() * (i + 1)); // j dans 0..i
        swap(a, i, j);
    }
}
```


- est facile à écrire
- est de complexité $O(N)$
- donne chacune des $N!$ permutations avec la même probabilité

complexité du problème du tri

à quelle vitesse peut-on espérer trier ?

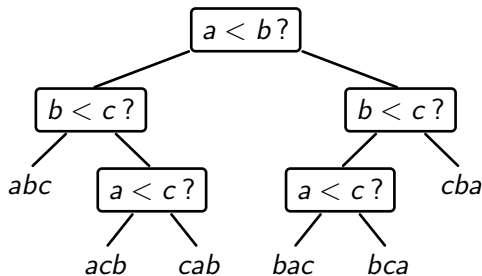
dit autrement : quelle est la meilleure complexité dans le pire des cas ?

on suppose que l'algorithme

- ne procède que par **comparaisons** entre éléments
- ne possède aucune information quant à la distribution

le coût sera justement le nombre de comparaisons effectuées

on peut trier trois valeurs a, b, c ainsi



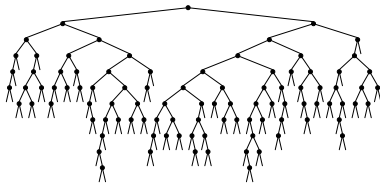
(un tel arbre, appelé *questionnaire*, représente l'algorithme)

ici, on fait 2 ou 3 comparaisons selon l'entrée

donc **3 comparaisons** dans le pire des cas

on se persuade facilement que c'est optimal pour $N = 3$:
on ne peut trier trois valeurs avec au plus deux comparaisons

un algorithme qui trie N valeurs peut être vu comme un arbre binaire où chaque nœud est un test et chaque feuille une permutation



- il y a **au moins** $N!$ feuilles
- la complexité dans le pire des cas est la **hauteur** de cet arbre

$$N! \leq \text{nombre de feuilles} \leq 2^{\text{hauteur}}$$

donc

$$\text{hauteur} \geq \log_2(N!) = \sum_{1 \leq i \leq N} \log_2(i) \sim N \log_2(N)$$

aucun algorithme, procédant par comparaisons uniquement,
ne peut trier N valeurs en effectuant toujours moins que

$$N \log N$$

comparaisons

le **tri par tas**, vu la semaine dernière, est optimal

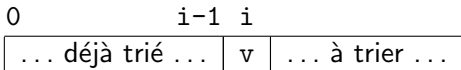
bien entendu, on peut faire mieux quand on possède une information supplémentaire sur les éléments

exemple : si les N éléments ne prennent que **deux valeurs** distinctes, alors on peut trier en $O(N)$

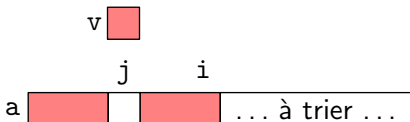
(cf exercices 86, 87 et 88 du poly)

tri par insertion

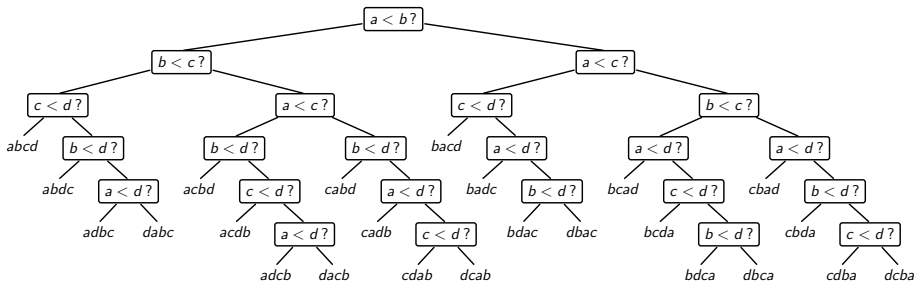
le plus naturel : on insère successivement chaque nouvel élément parmi les éléments déjà triés



```
static void insertionSort(int[] a) {  
    for (int i = 1; i < a.length; i++) {  
        int v = a[i], j = i;  
        // on décale les éléments vers la droite  
        for (; 0 < j && v < a[j-1]; j--) a[j] = a[j-1];  
        // jusqu'à avoir trouvé la place j de v  
        a[j] = v;  
    }  
}
```



a	b	c	d
---	---	---	---



(hauteur $6 > \lceil \log_2(24) \rceil$)

dans le **pire** des cas, l'insertion de $a[i]$ demande i comparaisons (tableau trié en ordre inverse), d'où un total

$$1 + 2 + \dots + N - 1 \sim \frac{N^2}{2}$$

dans le **meilleur** des cas, l'insertion de $a[i]$ demande une seule comparaison (le tableau est déjà trié), d'où un total

$$N$$

	meilleur cas	moyenne	pire cas
comparaisons	N	$N^2/4$	$N^2/2$
affectations	N	$N^2/4$	$N^2/2$

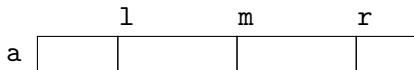
- + linéaire sur un tableau déjà trié
- quadratique dans le pire des cas
 $N = 10^5 \Rightarrow$ plusieurs milliards d'opérations

tri fusion

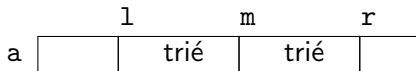
1. couper le tableau en deux moitiés égales
2. les trier récursivement
3. fusionner les résultats

il faut généraliser au tri de $a[1..r[$

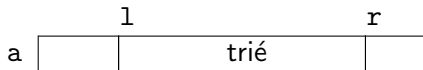
1. couper le tableau en deux moitiés égales $m = \lfloor \frac{1+r}{2} \rfloor$

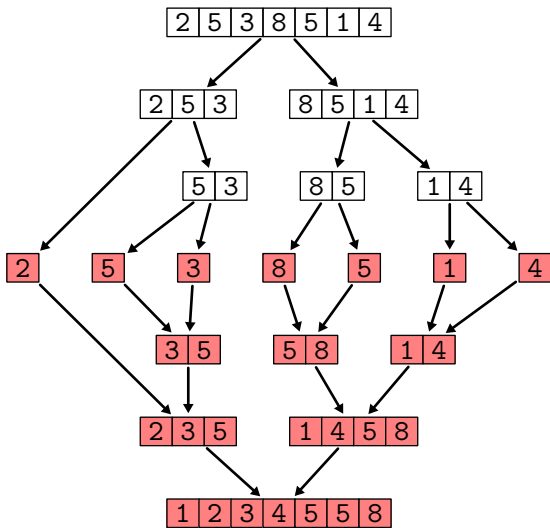


2. trier récursivement $a[1..m[$ et $a[m..r[$



3. fusionner les résultats





il est extrêmement difficile de réaliser la fusion **en place**

on va utiliser un second tableau

```
static void mergesort(int[] a) {  
    mergesortrec(a, new int[a.length], 0, a.length);  
}
```

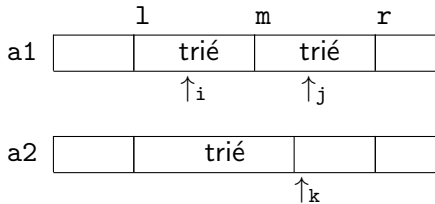
trier $a[l..r[$ en utilisant le tableau auxiliaire tmp

```
static void mergesortrec(int[] a, int[] tmp, int l, int r){
    if (l >= r-1) return; // au plus un élément
    int m = l + (r - l) / 2;
    mergesortrec(a, tmp, l, m);
    mergesortrec(a, tmp, m, r);
    for (int i = l; i < r; i++) tmp[i] = a[i];
    merge(tmp, a, l, m, r);
}
```

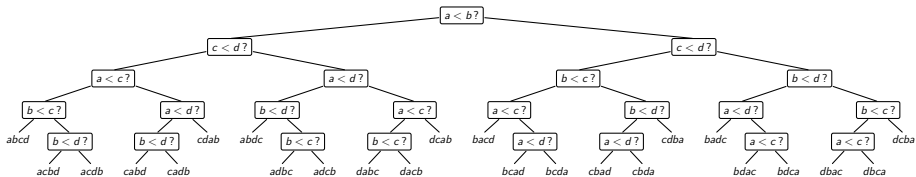
(note : $(l + r) / 2$ pourrait provoquer un débordement arithmétique)

fusionne $a1[1..m[$ et $a1[m..r[$ dans $a2[1..r[$

```
static void merge(int[] a1, int[] a2, int l, int m, int r){
    int i = l, j = m;
    for (int k = l; k < r; k++)
        if (i < m && (j == r || a1[i] <= a1[j]))
            a2[k] = a1[i++];
        else
            a2[k] = a1[j++];
}
```



a	b	c	d
-----	-----	-----	-----



(hauteur $5 = \lceil \log_2(24) \rceil$)

soit C_N le nombre de comparaisons pour trier N éléments

on a

$$\begin{cases} C_0 = C_1 = 0 \\ C_N = C_{\lfloor \frac{N}{2} \rfloor} + C_{\lceil \frac{N}{2} \rceil} + f_N \end{cases}$$

où f_N est le coût de la fusion

supposons le pire des cas ($f_N = N$) et $N = 2^n$ pour simplifier les calculs

il vient

$$C_{2^n} = C_{2^{n-1}} + C_{2^{n-1}} + 2^n$$

$$\frac{C_{2^n}}{2^n} = \frac{C_{2^{n-1}}}{2^{n-1}} + 1$$

$$= \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1$$

$$= n$$

c'est-à-dire

$$C_N = N \log_2(N)$$

	meilleur cas	moyenne	pire cas
comparaisons	$\frac{1}{2} N \log N$	$N \log N$	$N \log N$
affectations	$2N \log N$	$2N \log N$	$2N \log N$

- + complexité optimale
- espace supplémentaire $O(N)$

- + s'applique facilement aux listes, **en place**
(cf TD de cette semaine)

les appels récursifs imbriqués occupent de l'espace sur la pile (cf amphi 1)

mais cet espace est en $O(\log N)$,

car $|r - 1|$ est divisé par deux à chaque fois

donc on ne provoquera pas `StackOverflowError`

- quand $r - 1$ devient très petit, faire un tri par insertion
- pas besoin de fusionner si $a[m-1] \leq a[m]$
(devient alors linéaire sur un tableau déjà trié)
- pour éviter la copie dans `tmp`,
échanger le rôle des deux tableaux à chaque fois
(cf exercice 83 du poly)

on a procédé de haut en bas (*top-down mergesort*)

on peut aussi procéder **de bas en haut** (*bottom-up mergesort*)

- soit en partant de segments de taille 1
- soit en partant de segments déjà triés (*natural mergesort*)

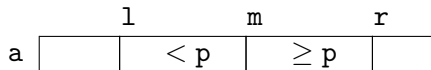
tri rapide

comment éviter l'espace auxiliaire

tout en conservant le principe *diviser pour régner* ?

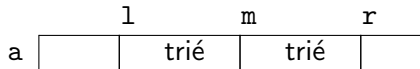
on conserve l'idée de trier $a[1..r]$

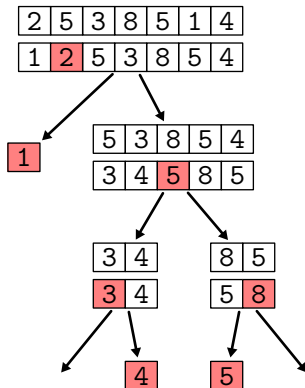
1. **réarranger** les éléments pour avoir



pour une certaine valeur p appelée **pivot**

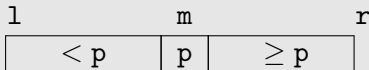
2. **trier récursivement** $a[1..m]$ et $a[m..r]$





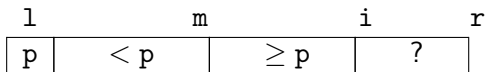
```
static void quicksort(int[] a) {
    quickrec(a, 0, a.length);
}
```

```
static void quickrec(int[] a, int l, int r) {
    if (l >= r-1) return; // au plus un élément
    int m = partition(a, l, r);
    quickrec(a, l, m);
    quickrec(a, m + 1, r);
}
```

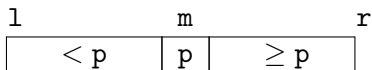


il ne reste plus qu'à écrire partition

on choisit arbitrairement $a[l]$ comme pivot



```
static int partition(int[] a, int l, int r) {
    int p = a[l], m = l;
    for (int i = l + 1; i < r; i++)
        if (a[i] < p)
            swap(a, i, ++m);
    swap(a, l, m);
    return m;
}
```



soit C_N le nombre de comparaisons pour trier N éléments

$$\left\{ \begin{array}{l} C_0 = C_1 = 0 \\ C_N = \underbrace{N-1}_{\text{partition}} + \underbrace{C_K}_{\text{à gauche}} + \underbrace{C_{N-1-K}}_{\text{à droite}} \end{array} \right.$$

pour un tableau déjà trié, $K = 0$

$$C_N = N - 1 + C_{N-1} \sim \frac{N^2}{2}$$

en moyenne, cependant, c'est optimal (preuve dans le poly)

	meilleur cas		moyenne	pire cas
comparaisons	$N \log N$		$2N \log N$	$N^2/2$
affectations		0	$2N \log N$	N^2

meilleur cas comparaisons = pivot au milieu

meilleur cas affectations = pivot à gauche = pire cas comparaisons

- + en place
- pire cas $O(N^2)$

le pire cas correspond à un pivot au bord

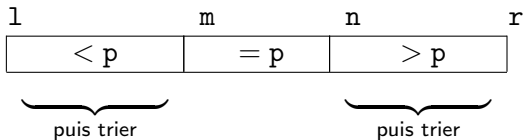
idée : prendre comme pivot un élément de $a[1..r]$ au hasard

encore plus simple : **mélanger avant de trier** !

```
static void quicksort(int[] a) {  
    KnuthShuffle.shuffle(a);  
    quickrec(a, 0, a.length);  
}
```


ne suffit cependant pas si beaucoup d'éléments égaux
(tous les éléments égaux \Rightarrow pivot forcément au bord)

solution : **partitionner en trois** (*3-way partition*)



(cf exercices 80 et 87 dans le poly)

comme pour le tri fusion,
les appels récursifs imbriqués occupent de l'espace sur la pile

mais ici cet espace peut être en $O(N)$
(on ne découpe plus systématiquement en deux moitiés égales)

provoque alors `StackOverflowError` !

on peut

- faire un appel récursif sur la plus petite des deux moitiés
- remplacer l'autre appel récursif par une boucle `while`

on retrouve alors une taille de pile en $O(\log N)$

(cf exercice 81 du poly)

- quand $r - 1$ devient très petit, faire un tri par insertion

tri par tas

```

static void moveDown(int[] a, int i, int x, int n) {
    while (true) {
        int j = 2 * i + 1;
        if (j >= n) break;
        if (j + 1 < n && a[j] < a[j + 1]) j++;
        if (a[j] <= x) break;
        a[i] = a[j]; i = j;
    }
    a[i] = x;
}

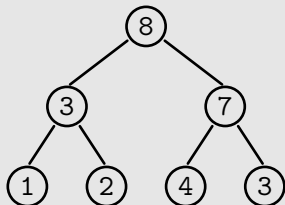
```

a 8 3 7 1 2 4 3

```

static void heapsort(int[] a) {
    int n = a.length;
    for (int k = n / 2 - 1; k >= 0; k--)
        moveDown(a, k, a[k], n);
    for (int k = n - 1; k >= 1; k--) {
        int v = a[k]; a[k] = a[0];
        moveDown(a, 0, v, k);
    }
}

```



le tri par tas est en $O(N \log N)$ dans le pire des cas (cf amphi 5)
et en place

pourquoi tous ces tris ?

ils ont chacun des avantages et des inconvénients

- en place ou non
- efficace sur un tableau (presque) déjà trié
- optimal dans le pire des cas
- stable

Définition

Un tri est **stable** s'il préserve l'ordre d'apparition des éléments égaux.

non significatif pour des éléments de type `int`

mais de manière générale on trie des objets d'un type E quelconque

```
class Sort<E extends Comparable<E>> { ... }
```

```
static<E extends Comparable<E>> void sort(E[] a) { ... }
```

un ensemble de films est donné par ordre chronologique

on trie selon la note moyenne des commentaires, avec un tri stable

résultat : à note égale, les films restent triés par ordre chronologique

(c'est une façon d'obtenir un ordre lexicographique)

	moyenne	pire cas	déjà trié	en place	stable
insertion	$O(N^2)$	$O(N^2)$	$O(N)$	oui	oui
fusion	$O(N \log N)$	$O(N \log N)$	$O(N)$	non	oui
rapide	$O(N \log N)$	$O(N^2)$	$O(N^2)$	oui	non
par tas	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	oui	non

et les constantes cachées dans les O peuvent faire une différence

fournit un grand nombre de méthodes `Arrays.sort(...)`

- tri **rapide** pour les types primitifs

```
void sort( char[] a);  
void sort( int[] a);  
void sort(double[] a);  
    ...
```

- tri **fusion** pour les objets

```
void sort(T[] a); // si T implémente Comparable<T>  
void sort(T[] a, Comparator<T> c);
```

la documentation garantit un tri stable

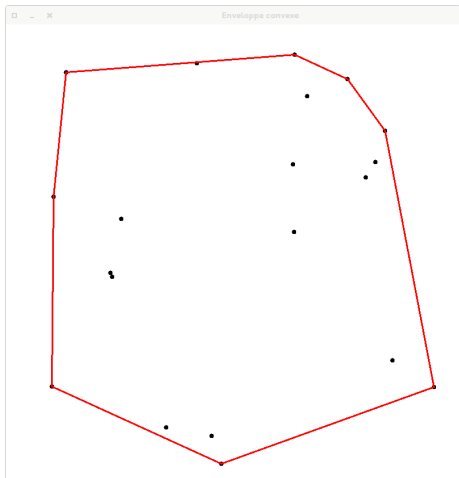
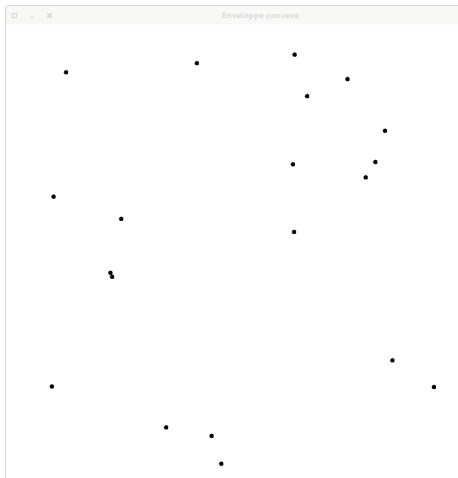
le tri ne sert pas qu'à organiser des données

c'est aussi un ingrédient de base de nombreux algorithmes

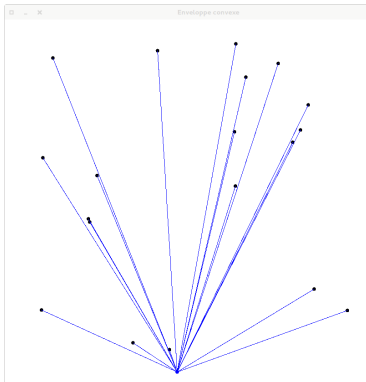
exemple : calcul de l'enveloppe convexe

entrée = N points dans le plan

sortie = points formant l'enveloppe convexe



- 1 déterminer le point p le plus bas
- 2 **trier** tous les points selon l'angle fait avec p



- 3 considérer les points dans cet ordre, en les ajoutant/retirant de l'enveloppe convexe

l'étape 1 est clairement en $O(N)$

l'étape 2 est en $O(N \log N)$ (tri de N valeurs)

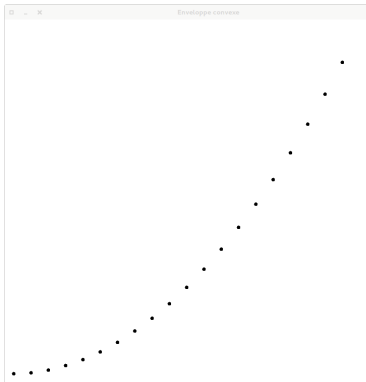
l'étape 3 est $O(N)$, car chaque point est ajouté une fois dans l'enveloppe, retiré au plus une fois de l'enveloppe

c'est donc une solution en $O(N \log N)$

on ne peut pas faire mieux

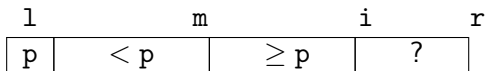
en effet, soient N entiers x_1, \dots, x_N

trouver l'enveloppe convexe des N points (x_i, x_i^2) revient à les trier



et on a vu que le tri ne peut être fait en moins de $N \log N$

quand on programme, il faut faire des petits dessins !



le tri fusion de listes, en place

application : une autre façon de trouver les mots les plus fréquents

Le → tour → du → monde → ...

à → à → à → à → ...

à(1678) → ah(4) → bien(13) → ...

de(2826) → à(1678) → le(1616) → la(1499) → ...

- **lire le poly**, chapitre 13. Tri

il y a des **exercices** dans le poly
suggestions : ex 84 p 157, ex 87 p 161

- **bloc 9** : graphes (1/2)