

IUT	Robert Schuman	
Institut universitaire de technologie		
Département <b>informatique</b>		
Université de Strasbourg		

# P4Z : Complexité Algorithmique

2019/20

## 1 Algorithmes de tris

Nous allons étudier quelques *algorithmes de tris* classiques pour introduire quelques problèmes élémentaires de complexité algorithmique dans un cadre accessible.

Le *problème du tri* est simple à comprendre : on dispose d'un ensemble d'objets munis de *clés* pour lesquelles on a une *relation d'ordre* et on se donne comme objectif de classer ces objets dans l'ordre des clés.

### Exemples :

- 1) liste de mp3 qu'on voudrait classer dans l'ordre alphabétique par rapport aux artistes ;
- 2) étudiants de S1 du DUT informatique qu'on veut classer par rapport à leurs résultats en M12 ;
- 3) une main de 18 cartes de tarot ;
- 4) plus prosaïquement : un tableau d'entiers qu'on veut classer dans l'ordre croissant...

### 1.1 Tri par Insertion (Insertion Sort)

Le principe est simple, c'est celui d'un classement de jeu de cartes : on reçoit des cartes au fur et à mesure et chaque nouvelle carte est insérée de manière à garder une main bien rangée. Voici l'algorithme pour un tableau  $A$  de taille  $n$  :

---

**triInsertion(A,n) :**

```

pour  $i = 1$  à  $n - 1$  faire
  clé =  $A[i]$ 
   $j = i - 1$ 
  tant que  $j \geq 0$  et  $A[j] > \text{clé}$  faire
     $A[j + 1] = A[j]$ 
     $j = j - 1$ 
  fin tant que
   $A[j + 1] = \text{clé}$ 
fin pour

```

---

Cet algorithme, bien que rudimentaire, est efficace sur de petits tableaux ainsi que sur des tableaux « partiellement triés ». C'est aussi un tri *stable*.

## 1.2 Tri Fusion (Merge Sort)

C'est un algorithme moins intuitif mais cependant très connu illustrant une technique algorithmique : « diviser et régner » (« divide and conquer »).

Division du problème : on trie les deux moitiés du tableau.

On combine alors les deux sous-tableaux obtenus en les « fusionnant » suivant un algorithme très simple.

Et comment trie-t-on les deux sous-tableaux ? On règne en appliquant récursivement la méthode ci-dessus et on s'arrête lorsqu'on arrive à un sous-tableau de taille 1 qui est trivialement trié !

Ci-après, le *sousTriFusion* qui triera le sous-tableau  $A[p, r[$  (le sous-tableau des éléments d'indice  $i$  vérifiant  $p \leq i < r$ ) :

---

**sousTriFusion(A,p,r) :**

```

si  $p < r - 1$  alors
   $q = \text{Ent}((p + r)/2)$  (partie entière)
  sousTriFusion( $A, p, q$ )
  sousTriFusion( $A, q, r$ )
  fusion( $A, p, q, r$ )
fin si

```

---

L'algorithme ci-dessus ne fait rien si la longueur est 1. Sinon il scinde puis fusionne les deux sous-tableaux après tri de manière récursive. Ci-après l'algorithme de fusion :

---

**fusion(A,p,q,r) :**
 $n_1 = q - p$  (nb de valeurs dans  $A[p, q[$ )

 $n_2 = r - q$  (nb de valeurs dans  $A[q, r[$ )
Créer des tableaux  $A_g$  et  $A_d$  de tailles respectives  $n_1$  et  $n_2$ Copier  $A[p, q[$  dans  $A_g$  et  $A[q, r[$  dans  $A_d$  $ind_g = 0$  $ind_d = 0$  $i = p$ **tant que**  $i < r$  **faire**  **si**  $ind_g == n_1$  **alors**     $A[i] = A_d[ind_d]$      $ind_d ++$   **sinon si**  $ind_d == n_2$  **alors**     $A[i] = A_g[ind_g]$      $ind_g ++$   **sinon si**  $A_g[ind_g] < A_d[ind_d]$  **alors**     $A[i] = A_g[ind_g]$      $ind_g ++$   **sinon**     $A[i] = A_d[ind_d]$      $ind_d ++$   **fin si**   $i ++$ **fin tant que**


---

Évidemment, la fonction de tri fusion d'un tableau  $A$  de taille  $n$  appelle *sousTriFusion* :

---

**triFusion(A,n) :**
 $sousTriFusion(A, 0, n)$ 


---

### 1.3 Tri Rapide (Quick Sort)

Un algorithme qui porte bien son nom et qui est très répandu. L'idée est encore une fois simple : on se choisit un élément particulier dans le tableau (le *pivot*) et on réorganise le tableau en plaçant à gauche du pivot tous les éléments plus petits que lui et à sa droite tous les éléments plus grands (on appelle cela la *partition*).

Ensuite, on applique la même méthode aux sous-tableaux de manière récursive.

Dans l'exemple ci-dessous, on donne l'algorithme sous sa forme bête et méchante, à savoir par exemple que le choix du pivot sera toujours le dernier élément du tableau à trier. L'algorithme *sousTriRapide* triera le sous-tableau  $A[p, r[$  (les indices  $i$  avec  $p \leq i < r$ ). L'algorithme *partition* retourne l'indice du pivot après avoir partitionné le sous-tableau.

---

**sousTriRapide(A, p, r) :**

```

si p < r-1 alors
  q = partition(A,p,r)
  sousTriRapide(A,p,q)
  sousTriRapide(A,q+1,r)
fin si

```

---



---

**partition(A, p, r) :**

```

pivot = A[r - 1]
i = p
pour j = p à r - 2 faire
  si A[j] ≤ pivot alors
    permuter A[i] et A[j]
    i++
  fin si
fin pour
permuter A[i] et A[r - 1]
retourner i

```

---

Évidemment, la fonction de tri rapide d'un tableau  $A$  de taille  $n$  appelle *sousTriRapide* :

---

**triRapide(A,n) :**

```

sousTriRapide(A, 0, n)

```

---

Autant se l'avouer tout de suite, il faudra réfléchir au rôle du pivot ou alors on risque d'avoir de mauvaises surprises.

Cet algorithme n'est pas stable mais est en place.

## 2 Exercices TD/TP

### Exercice 1

Vous avez toujours rêvé d'incarner un ordinateur (mais si !) et vous allez donc, à la main, tester les trois algorithmes sur les listes de nombres suivantes en donnant chaque étape et en comptant le nombre de comparaisons et d'écritures dans des tableaux ainsi que le nombre de fusions ou de partitions pour les tris concernés :

$$(3, 2, 5, 1, 4) \quad (3, 2, 8, 4, 1, 6, 7, 5).$$

### Exercice 2

Combien d'écritures et de comparaisons sur les tableaux ainsi que le nombre de fusions ou de partitions pour les tris concernés :

$$\{2^n, 2^{n-1}, 2^{n-2}, \dots, 2, 1\} \quad \text{et} \quad \{1, 2, \dots, 2^{n-1}, 2^n\}.$$

### Exercice 3

On s'intéresse ici aux tableaux d'entiers de taille  $n$ . En C, coder les trois tris :

- 1) `void triInsertion(long* A, size_t n);`
- 2) `void triFusion(long* A, size_t n);`
- 3) `void triRapide(long* A, size_t n);`

Ensuite, étudier ces tris :

- a) Tester les performances grâce aux techniques de sioux que M. Gossa a présentées.
- b) Compter le nombre d'écritures/d'échanges/de comparaisons/d'appels à fusion/d'appels à partition.
- c) Trouver quels types de tableaux minimisent le temps de calcul pour chaque tri.
- d) Trouver quels types de tableaux maximisent le temps de calcul pour chaque tri.

<b>Date du premier rendu : vendredi 31 janvier 23h59 au plus tard</b>
---

Les rendus se feront dans un dossier nommé TP1 dans votre le dépôt P4z (cloné à partir de celui de M. Gossa).

Les fonctions seront codées dans un fichier `tris.c` (avec le fichier header correspondant) avec bien sûr un fichier `main.c` ainsi qu'un fichier `makefile`. Il est conseillé de se faire un fichier de fonctions utilitaires permettant d'afficher des tableaux, de créer divers types de tableaux aléatoires, ...

La structure des fichiers sera amenée à évoluer lors des TP suivants.

**Exercice 4**

Pour voir plus loin et mieux :

- 1) Tester le tri rapide sur des tableaux très longs de nombres déjà triés en ordre décroissant. Le point faible est le choix du pivot. Essayer de régler le problème par un procédé aléatoire.
- 2) Le tri rapide et le tri fusion ne sont pas très performants sur de petits tableaux, contrairement au tri par insertion. En tout cas, cela se vérifie! Améliorer les tris rapides et tris fusions.
- 3) Tester le tri rapide sur des tableaux très longs de nombres qui ne peuvent prendre qu'un nombre réduit de valeurs. Comment résoudre le problème?
- 4) Faire des concours de tris et ne pas hésiter à en essayer d'autres (sélection, à bulles, tas, Shell, dénombrement, Gnome, peigne, ...).
- 5) Trouver l'algorithme de tri ultime.