Tri rapide (*Quicksort*)

Lundi 12 Mars 2018

Michael FRANÇOIS

francois@esiea.fr

Tri rapide (*quicksort*)

- Le tri rapide (en anglais *quicksort*) est un algorithme de tri créé en 1960 par **T. HOARE**, étudiant en visite à l'université d'État de Moscou.
  - Il avait en effet besoin d'un algorithme capable de trier des mots devant être traduits, pour une correspondance à un dictionnaire Russe-Anglais déjà existant, stocké sur une bande magnétique.

- Le tri rapide comme le tri fusion est fondé sur le paradigme diviser-pour-régner. Il est généralement utilisé sur des tableaux ou même des listes.

- C'est un algorithme de tri dont le temps d'exécution sur un tableau constitué de n nombres est de $O(n^2)$ dans le pire des cas.

- Cependant, cet algorithme reste souvent le meilleur choix en pratique, à cause de son efficacité remarquable en moyenne : temps d'exécution espéré en $O(n \log n)$.

- Cet algorithme correspond à la méthode de tri de tableaux la plus rapide pour des données bien **mélangées** ou réparties de manière **aléatoire** $\Longrightarrow$ (complexité $O(n \log n)$).

- Mais dans le cas où le tableau de données est trié ou presque trié, cet algorithme est théoriquement et pratiquement très lent $\Longrightarrow$ (complexité $O(n^2)$).

- Cet algorithme reste naturellement récursif.

Description du tri rapide

Le principe général du tri rapide est simple :

- Au départ on choisit "au hasard" un élément du tableau comme étant le **pivot** et on **partitionne** le tableau autour de ce pivot. Cette étape engendre deux sous-tableaux appelés :
  sous-tableau gauche et sous-tableau droit.
- On continue de manière récursive en appelant la même fonction sur chacun des deux sous-tableaux gauche et droit, jusqu'à ce que le tableau en entier soit trié.

En clair : pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété généralement récursivement, jusqu'à ce que l'ensemble du tableau soit trié.

Voici les trois étapes du processus diviser-pour-régner employé pour trier un tableau typique `TAB[g...d]` :

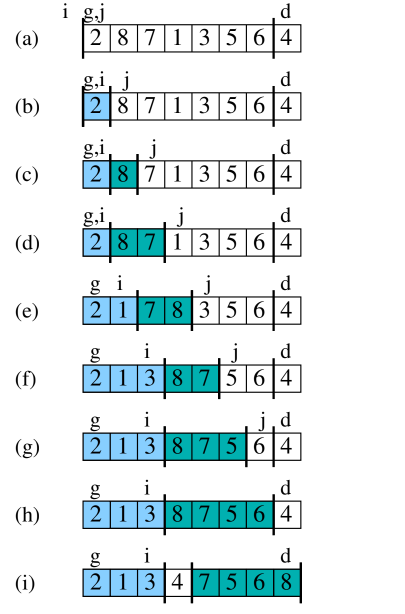
1. **Diviser :** le tableau `TAB[g...d]` est partitionné (réarrangé) en deux sous-tableaux `TAB[g...m-1]` et `TAB[m+1...d]` tels que chaque élément de `TAB[g...m-1]` soit inférieur ou égal à `TAB[m]`, qui lui même est inférieur ou égal à chaque élément de `TAB[m+1...d]`. L'indice *m* est bien-sur celui du pivot.
2. **Régner :** les deux sous-tableaux `TAB[g...m-1]` et `TAB[m+1...d]` sont triés par des appels récursifs au tri rapide.
3. **Combiner :** les sous-tableaux étant triés, aucun travail n'est nécessaire pour les combiner : le tableau `TAB[g...d]` est maintenant globalement trié.

La procédure suivante implémente le tri rapide :

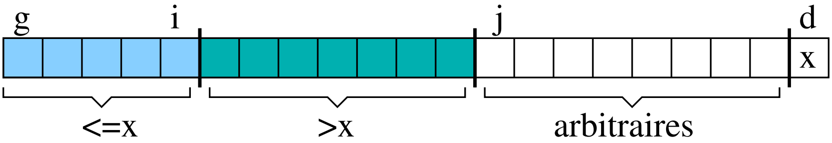
```
TRI_RAPIDE(TAB, g, d)                |
DEBUT                                |
 Si g < d alors                      |
    m = PARTITION(TAB, g, d)         | Pour trier un tableau TAB d'entiers,
    TRI_RAPIDE(TAB, g, m-1)          | l'appel initial est :
    TRI_RAPIDE(TAB, m+1, d)          | TRI_RAPIDE(TAB, 1, TAB.longueur)
 fin Si                              |
FIN                                  |

PARTITION(TAB, g, d)                 |
DEBUT                                |
 x = TAB[d] /*On choisit le pivot*/|
 i = g-1                             | Le point principal de l'algorithme
 Pour j=g à d-1                      | est la fonction PARTITION, qui
    Si TAB[j]<=x                     | réarrange le sous-tableau TAB[g...d]
       i=i+1                         | sur place.
       permuter TAB[i] <--> TAB[j]   | x=TAB[d] est choisi comme pivot, autour
    fin Si                           | duquel se fera le partitionnement de TAB.
 fin Pour                            | Temps d'exécution : O(n) où n = d-g+1
 permuter TAB[i+1] <--> TAB[d]       | Cette procédure PARTITION est celle de :
 Retourner i+1                       | N. LOMUTO. Celle de T. HOARE sera étudiée
FIN                                  | à la fin de cours et également en TD.
```

Fonctionnement de la fonction `PARTITION` sur un exemple à 8 éléments :



L'élément de `TAB[d]` devient le pivot x. Quand la procédure est exécutée, le tableau est partitionné en 4 régions. Par exemple les éléments en bleu clair sont tous dans la 1ère partition, avec des valeurs pas plus grande que x. Les éléments en vert foncé sont dans la seconde partition (ou partition des grandes valeurs), avec des valeurs supérieures à x. La 3ème partition accueille les éléments pas encore traités et la 4ème contient le pivot x. Les deux partitions grossissent au fur et à mesure et la fin (i), l'élément pivot est permuté de façon à aller entre les deux partitions.



Pour tout indice k :

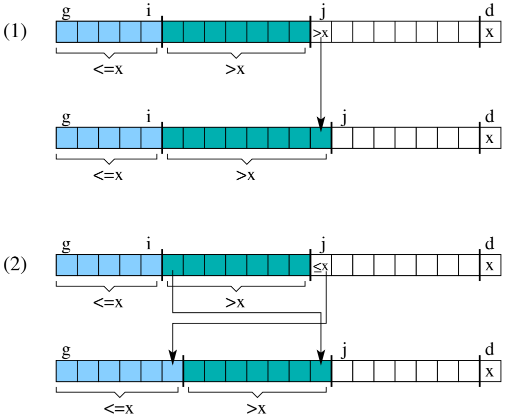
- si $g \leq k \leq i$, alors `TAB[k]` $\leq$ `x` (Partition des petites valeurs)
- si $i + 1 \leq k \leq j - 1$, alors `TAB[k]` $>$ `x` (Partition des grandes valeurs)
- si $k = d$, alors `TAB[k] = x` (place du pivot)

Remarque : ces propriétés définissent un invariant de boucle.

Cet invariant de boucle est vrai avant, pendant et après les itérations :

Initialisation : avant la 1ère itération, $i = g - 1$ et $j = g$. Il n'y a pas de valeurs entre g et i, ni entre $i + 1$ et $j - 1$. Ceci satisfait trivialement les deux 1ères conditions de l'invariant de boucle.

Conservation : il y a deux cas à considérer ici :



(1) Si `TAB[j]>x`, la seule action consiste à incrémenter j, ce qui conserve l'invariant de boucle. (2) Si `TAB[j]`$\leq$`x`, i est incrémenté, `TAB[i]` et `TAB[j]` sont échangés, puis j est incrémenté. Ici également l'invariant de boucle est respecté.

Terminaison : à la fin (*i.e.* $j = d$), les valeurs du tableaux sont partitionnées en 3 ensembles : les valeurs $\leq x$, les valeurs $> x$ et un singleton contenant le pivot x.

Performances du Tri rapide

• Le temps d'exécution du tri rapide est lié au caractère équilibré ou non du partitionnement. En cas de partitionnement équilibré, l'algorithme s'exécute asymptotiquement aussi vite que le tri-fusion :

- Partitionnement (non-équilibré) dans le cas le plus défavorable $\longrightarrow$ $O(n^2)$.
- Partitionnement (équilibré) dans le cas le plus favorable $\longrightarrow$ $O(n \log n)$.

• Un partitionnement équilibré à chaque niveau de la récursivité engendre un algorithme plus rapide asymptotiquement.

Partition autour de 0

- Voici une variante du problème de partition :
  - On a un tableau `TAB` d'entiers de taille n éléments.
  - Ce tableau n'est composé que d'éléments de valeurs 0 ou 1 (mais on ne le sait pas au départ).
  - On désire obtenir le tableau `TAB` partitionné autour de 0.
- Au départ `TAB` est :

0	1	1	0	1	1	0	1	0	0

Et on désire obtenir :

0	0	0	0	0	1	1	1	1	1

Partitionnement autour de la valeur 0.

```
-----------------------------------------------------------
PARTITION_ZERO(TAB, n)  //Stratégie Gauche-Droite
DEBUT
 k <-- 0
 Pour i de 0 à n-1 faire
    Si TAB[i]<=0 alors
        temp <-- TAB[k]
        TAB[k] <-- TAB[i]
        TAB[i] <-- temp
        k <-- k+1
    fin SI
 fin Pour
 Retourner k
FIN
-----------------------------------------------------------
```

Explications :

• Lorsqu'on rencontre un élément ≤ 0 : on le place à la première place libre (k) puis on incrémente k de 1 pour préparer la place au prochain élément à permuter.

• Ainsi, au début si on rencontre un élément ≤ 0, il s'échange avec lui même (puisque $k = 0$ initialement), c'est un défaut apparent mais en fait nécessaire.

• Donc k représente au final le nombre d'éléments rencontrés qui sont ≤ 0.

• L'algorithme précédent avait une stratégie gauche-Droite.
Comment mettre en place la stratégie Droite-Gauche ????

Partitionnement autour de la valeur 0.

```
-----------------------------------------------------------
PARTITION_ZERO(TAB, n)  //Stratégie Droite-Gauche
DEBUT
 k <-- n-1
 Pour i de n-1 à 0 faire
    Si TAB[i]>0 alors
        temp <-- TAB[k]
        TAB[k] <-- TAB[i]
        TAB[i] <-- temp
        k <-- k-1
    fin SI
 fin Pour
 Retourner k
FIN
-----------------------------------------------------------
```

Recherche du minimum/maximum dans un tableau

Recherche du minimum/maximum dans un tableau

- Pour la recherche du minimum/maximum (extrema) d'un tableau de n éléments, il existe un seul algorithme avec plusieurs variantes possibles.

- Si on ne dispose pas d'informations spécifiques, on doit faire $n-1$ comparaisons pour trouver le maximum (resp. minimum) d'un tableau à n éléments.

- Et si on recherche le minimum et le maximum à la fois ? Car il peut exister des situations où on désire trouver les deux à la fois : par exemple cadrer un ensemble de points, pour cela il faut chercher les abscisses (resp. ordonnées) minimales et maximales.

Exemple : dans le cas où on recherche uniquement la **valeur** du minimum dans le tableau :

```
----------------------------------------------------------
MINIMUM_VAL(TAB, n)
DEBUT
 Min <-- TAB[0]
 Pour i allant de 1 à n-1 inclus faire
     Si TAB[i] < Min alors
        Min <-- TAB[i]
     fin SI
 fin Pour
 Retourner Min
FIN
----------------------------------------------------------
```

Même principe que pour le maximum.

Exemple : dans le cas où on recherche l'**indice** du minimum dans le tableau :

```
--------------------------------------------------------
MINIMUM_IND(TAB, n)
DEBUT
 IndMin <-- 0
 Pour i allant de 1 à n-1 inclus faire
     Si TAB[i] < TAB[IndMin] alors
        IndMin <-- i
     fin SI
 fin Pour
 Retourner IndMin
FIN
--------------------------------------------------------
```

Même principe que pour le maximum.

Exemple : dans le cas où on recherche le minimum et le maximum dans le tableau : (version naïve)

```
-------------------------------------------------------------
MIN_MAX(TAB, n)
DEBUT
 Si TAB[0] < TAB[1] alors
    MIN <-- TAB[0] MAX <-- TAB[1]
 Sinon
    MIN <-- TAB[1] MAX <-- TAB[0]
 fin SI
 Pour i allant de 2 à n-1 inclus faire
     Si TAB[i] < MIN alors
        MIN <-- TAB[i]
     Sinon SI TAB[i] > MAX alors
        MAX <-- TAB[i]
     fin Si
 fin Pour
 Retourner MIN, MAX
FIN
-------------------------------------------------------------
```

Que donne la méthode diviser-pour-régner sur ce problème ?

En utilisant la méthode diviser-pour-régner :

```
----------------------------------------------------------
MIN_MAX_DPR(TAB, n)
DEBUT
 Définir TAB1 et TAB2 deux sous-tableaux de TAB
 Calculer MIN1 et MAX1 pour TAB1
 Calculer MIN2 et MAX2 pour TAB2
 Combiner pour trouver le min. et max. globaux
   MING=min(MIN1, MIN2)
   MAXG=max(MAX1, MAX2)
 Retourner MING, MAXG
FIN
----------------------------------------------------------
```

- On considère le tableau suivant :
TAB = [5, 18, 27, 7, 6, 12, 11, 15, 3, 10, 17, 13]

- On applique une fois l'instruction `PARTITION_LOMUTO(TAB, 0, 11)`.
Quel sera le nouveau arrangement de TAB ?

```
fonction PARTITION_LOMUTO(TAB, g, d)
    x ⟵ TAB[d] /*Le pivot*/
    i ⟵ g-1
    pour j allant de g à d-1 inclus faire
        si (TAB[j]≤x) alors
            i ⟵ i+1
            permuter TAB[i] <-> TAB[j]
        fin
    fin
    permuter TAB[i+1] <-> TAB[d] /*On ramène le pivot entre les deux partitions*/
    Retourner i+1
```

Solution :
TAB $= \{5, 7, 6, 12, 11, 3, 10, 13, 18, 27, 17, 15\}$

- On considère le tableau suivant :
TAB = [5, 18, 27, 7, 6, 12, 11, 15, 3, 10, 17, 13]

- On applique une fois l'instruction `PARTITION_HOARE(TAB, 0, 11)`.
Quel sera le nouveau arrangement de TAB ?

```
fonction PARTITION_HOARE(TAB, g, d)
    x ←— TAB[g] /*Le pivot*/
    i ←— g-1
    j ←— d+1
    tant que VRAI faire
        répéter
            j ←— j-1
        tant que (TAB[j]>x);
        répéter
            i ←— i+1
        tant que (TAB[i]<x);
        si (i<j) alors
            permuter TAB[i] <-> TAB[j]
        fin
        sinon
            Retourner j
        fin
    fin
```

Solution :
TAB $= \{3, 18, 27, 7, 6, 12, 11, 15, 5, 10, 17, 13\}$

Bibliographie

- R. ERRA, "Cours d'informatique", 1A-S2 2013-2014 ESIEA.
- Cormen, Leiserson, Rivest, Stein "Algorithmique", 3ème éd. DUNOD.