Master Sciences, Technologies, Santé
Mention Mathématiques, spécialité Enseignement des mathématiques

MHT063

Algorithmique et graphes, thèmes du second degré

ALGORITHMES DE TRI

Éric SOPENA
Eric.Sopena@labri.fr

SOMMAIRE

Chapitre 1. Introduction

L'organisation de la mémoire est un problème essentiel quand on programme un tri : si la taille de l'ensemble des objets à trier permet de travailler en mémoire centrale, on parle de *tri interne*. Dans le cas contraire, on parle de *tri externe* (les objets à trier sont alors regroupés dans un fichier).

De nombreux algorithmes existent : l'analyse et la comparaison de ceux-ci sont donc indispensables. Il existe des résultats d'optimalité (borne inférieure pour la complexité du problème de tri et existence d'algorithmes atteignant cette borne).

L'étude des algorithmes de tri représente une étape indispensable dans l'acquisition d'une culture informatique. Ces algorithmes permettent en effet d'illustrer de façon significative de nombreux problèmes rencontrés en Informatique. Cela étant, il est bien évidemment très rare que l'on soit amené à programmer un algorithme de tri : des outils sont en effet généralement offerts par les systèmes d'exploitation voire par certains langages. Lorsque l'on est tout de même amené à faire appel à un tel algorithme, on se contente de choisir un algorithme existant (d'où l'importance d'une certaine culture).

Dans la suite, nous illustrerons les algorithmes de tri interne à l'aide d'un tableau d'objets de type `TInfo` et les algorithmes de tri externe seront donnés dans leur version interne. Nous utiliserons donc pour cela le type suivant :

```
constante   CMAX = …
type        TTableau = tableau de CMAX TInfo
```

On supposera bien sûr que le type `TInfo` est muni d'une relation d'ordre. Pour les algorithmes de tri interne, le tableau sera trié « sur lui-même », c'est-à-dire sans utiliser d'espace supplémentaire (excepté un emplacement permettant d'effectuer des échanges). Les algorithmes de tris externes seront simulés en interne à l'aide de plusieurs tableaux (correspondant aux différents fichiers).

On utilisera des tableaux « de taille variable », représentés ainsi :

```
variables   tab : TTableau
            nbElem : entier naturel
```

où la variable `nbElem` correspond au nombre d'éléments effectivement présents dans le tableau `tab`.

Pour chaque algorithme de tri présenté, on s'intéressera à la *complexité en temps*, déclinée en deux paramètres : nombre de comparaisons et nombre de déplacements d'objets (important lorsqu'on trie de « gros » objets…).

Chapitre 2. Les méthodes de tri simples

Ces méthodes peuvent être regroupées en deux catégories : par sélection (on recherche l'élément devant figurer dans un emplacement donné) ou par insertion (à chaque étape, un élément est inséré parmi un sous-ensemble d'éléments déjà triés). Les performances obtenues sont assez médiocres en termes de complexité, mais ces tris simples sont aisés à mettre en œuvre et peuvent être utilisés lorsque le nombre d'informations à trier reste peu élevé.

Les opérations d'échange d'éléments du tableau sont réalisées par l'action suivante :

```
Action Echange ( ES tab : TTableau ; E i :entier ; E j : entier )
# cette action échange dans le tableau tab les éléments en position
# i et j
variable    temp : TInfo
début
   temp ← tab[i]
   tab[i] ← tab[j]
   tab[j] ← temp
fin
```

2.1. Sélection ordinaire

L'algorithme consiste à déterminer successivement l'élément devant se retrouver en 1ère position, 2ème position, etc., c'est-à-dire le plus petit, puis le plus petit des restants, et ainsi de suite.

On obtient l'algorithme suivant :

```
Action Tri_Sélection ( ES : tab : TTableau ; E nbElem : entier )
# cette action trie le tableau tab à nbElem éléments par insertion
variables   i, pos, posPlusPetit : entiers
début
       # on place les éléments adéquats en position 0, 1, 2, etc.
   Pour pos de 0 à nbElem - 2
   faire
            # on cherche le plus petit élément entre pos+1 et nbElem-1
        posPlusPetit ← pos
        Pour i de pos + 1 à nbElem - 1
        faire
            Si ( tab[i] < tab[posPlusPetit] )
            alors posPlusPetit ← i
        fin_pour
            # on range l'élément à sa place
        Echanger (tab, pos, posPlusPetit)
   fin_pour
fin
```

L'inconvénient majeur de cet algorithme est que de nombreuses comparaisons sont effectuées plusieurs fois…

Plus précisément, la complexité de cet algorithme est la suivante :

- déplacements : chaque élément placé l'est définitivement, ce qui, par échange, correspond à 3(n-1) déplacements, d'où une complexité en $\Theta(n)$.
- comparaisons : le nombre de comparaisons est (n-1) + (n-2) + … + 1 = n(n-1)/2, d'où une complexité en $\Theta(n^2)$.

La complexité de cet algorithme est donc en $\Theta(n^2)$.

2.2. Tri-bulle (ou *bubble sort*)

Le principe consiste à déplacer les petits éléments vers le début du tableau et les grands vers la fin du tableau en effectuant des échanges successifs. Le tableau est parcouru de droite à gauche ; ainsi, à l'issue d'un parcours, le plus petit élément (bulle) est amené en position correcte. Le tableau sera trié lorsqu'aucun échange ne sera plus nécessaire (un parcours du tableau sans découverte d'incompatibilité…).

L'algorithme est le suivant :

```
Action Tri_Bulle ( ES tab : TTableau ; E nbElem : entier )

# cette action trie le tableau tab à nbElem éléments par la méthode
# du tri-bulle (Bubble-sort)

variables   i, j : entiers
            fini : booléen

début
  i ← 0

  répéter
       fini ← Vrai

            # on parcourt le tableau de droite à gauche, des positions
            # nbElem – 1 à i+1
       pour j de nbElem – 1 à i + 1 par pas de -1
       faire    Si ( tab[j] < tab[j-1] )
                alors          # la bulle remonte
                          Echanger (T, j, j-1 )
                               # il faudra refaire un parcours
                          fini ← Faux
                fin_si
       fin_pour

       i ← i+1

            # on s'arrête à l'issue d'un parcours sans échanges ou
            # lorsqu'il ne reste plus qu'un élément à traiter
  Jusqu'à ( fini ou (i = nbElem - 2) )
fin
```

Au premier tour, le plus petit élément se retrouve « tout à gauche », et ainsi de suite. Si on effectue un corps de boucle sans aucun changement, cela signifie que les éléments sont déjà triés : le booléen `fini` reste à `Vrai` et on quitte la structure.

Intuitivement, le coût de cet algorithme provient du fait qu'un élément atteint sa place finale en passant par toutes les places intermédiaires (nombreux échanges), voire plus lorsqu'il « part à droite » avant de repartir vers la gauche...

On a ainsi :

- déplacements : dans le pire des cas (tableau trié inverse), le nombre d'échanges est de (n-1) + (n-2) + … + 1 = n(n-1)/2, soit une complexité au pire de $\Theta(n^2)$. Dans le meilleur des cas (tableau déjà trié), on n'effectue aucun déplacement (complexité en $\Theta(1)$).
- comparaisons : le nombre de comparaisons au pire est de (n-1) + (n-2) + … + 1 = n(n-1)/2, d'où une complexité au pire en $\Theta(n^2)$. Dans le meilleur des cas, on effectue n-1 comparaisons (pour se rendre compte que le tableau est trié), soit une complexité de $\Theta(n)$ au mieux.

La complexité de cet algorithme est donc au pire en $\Theta(n^2)$ et au mieux en $\Theta(n)$. On peut montrer qu'elle est également en $\Theta(n^2)$ en moyenne.

2.3. Insertion séquentielle

Cet algorithme procède par étapes : à la $i^{ième}$ étape, on insère l'élément `tab[i]` à sa place parmi les `i-1` premiers éléments déjà triés (c'est l'algorithme du *joueur de cartes*).

On obtient alors :

```
Action Tri_Insertion ( ES tab : Ttableau ; E nbElem : entier)

# cette action trie le tableau tab à nbElem éléments par insertion
# (méthode du joueur de cartes)

variables   pos, i, elem : entiers
            trouvé : booléen

début
       # on va insérer les éléments d'indice 1, 2, … en bonne position
       # dans la partie gauche du tableau
  pour i de 1 à nbElem - 1
  faire
            # on sauvegarde tab[i]
       elem ← tab[i]

            # on cherche la position de tab[i] vers sa gauche en
            # décalant les éléments plus grands
            # dès qu'on trouve un « petit », trouvé passe à Vrai
       pos ← i- 1
       trouvé ← Faux
       tantque ( (non Trouvé) et (pos >= 0) )
       faire     si ( tab[pos] > elem )
                 alors     tab[pos+1] ← tab[pos]
                           pos ← pos - 1
                 sinon     trouvé ← Vrai
                 fin_si
       fin_tantque

            # on range Elem à sa place
       tab[pos+1] ← elem

  fin_pour
fin
```

Concernant la complexité, nous avons alors :

- déplacements : dans le meilleur des cas (tableau déjà trié), on effectue 2n déplacements (sauvegarde inutile de `tab[i]`) (complexité en $\Theta(n)$). Dans le pire des cas (tableau trié inverse), le nombre de déplacements est de n+(n-1) + (n-2) + … + 1 = n(n+1)/2, soit une complexité au pire de $\Theta(n^2)$.
- comparaisons : le nombre de comparaisons au pire est de (n-1) + (n-2) + … + 1 = n(n-1)/2, d'où une complexité au pire en $\Theta(n^2)$. Dans le meilleur des cas, on effectue n-1 comparaisons (pour se rendre compte que le tableau est trié), soit une complexité de $\Theta(n)$ au mieux.

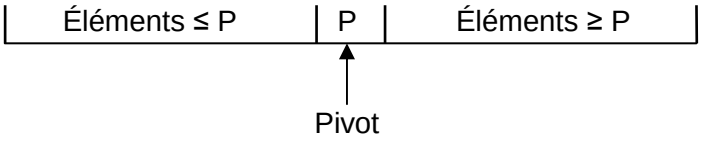
La complexité de cet algorithme est donc au pire en $\Theta(n^2)$ et au mieux en $\Theta(n)$. On peut montrer qu'elle est également en $\Theta(n^2)$ en moyenne.

2.4. Insertion dichotomique

On peut améliorer l'algorithme précédent en effectuant la recherche de la position de `Elem` non plus séquentiellement, mais par dichotomie : il y a bien sûr toujours autant de décalages, mais le nombre de comparaisons nécessaires diminue (en $\Theta(\log n)$). La complexité en moyenne reste donc en $\Theta(n^2)$, du fait des décalages nécessaires.

Chapitre 3. Le tri rapide : Quicksort

L'idée consiste à partitionner (ou réorganiser) le tableau de façon à obtenir la configuration suivante :



On peut ainsi assurer que l'élément `P` se trouve à sa place. On va ensuite trier, récursivement, les deux portions de tableau (éléments plus petits que `P`, éléments plus grands que `P`) selon le même principe. Lorsque ces deux portions sont triées, on peut alors affirmer que la totalité du tableau est triée.

Choix du pivot : n'ayant aucune connaissance *a priori* de la répartition des éléments à trier, on prendra le premier élément comme `Pivot`, et on répartira les éléments restants en deux classes : les « petits » et les « grands ». L'algorithme récursif correspondant sera ainsi amené à travailler sur des portions de tableau ; on utilisera donc deux paramètres d'entrée, les indices `deb` et `fin`, permettant de délimiter cette portion.

On obtient ainsi l'algorithme suivant :

```
Action Tri_Rapide ( ES tab : TTableau ; E deb, fin : entiers )
# cette action trie la portion du tableau tab comprise entre les indices
# deb et fin en utilisant la méthode de tri rapide (quicksort)
variables   posPivot : entier
début
   Si ( fin > deb )        # si la portion contient plus d'un élément
   Alors
            # on partitionne, posPivot reçoit la position du pivot
        Partitionner (tab, deb, fin, posPivot )
            # on trie les éléments plus petits que le pivot
        Tri_Rapide (tab, deb, posPivot-1 )
            # on trie les éléments plus grands que le pivot
        Tri_Rapide (tab, posPivot+1, fin )
   fin_si
fin
```

Le tri d'un tableau `tab` à `nbElem` éléments s'effectue alors par l'appel :

```
Tri_Rapide ( tab, 0, nbElem-1 )
```

Il reste maintenant à écrire l'action `Partitionner`. Le quatrième paramètre (de sortie) `posPivot` permet de retourner la position finale du pivot, à l'issue du partitionnement.

Le principe est le suivant :

- le premier élément de la portion (en position `deb`) sera le pivot,
- on cherche, de droite à gauche (à l'aide d'un indice `droite`), un élément plus petit que le pivot ; lorsqu'on le trouve, on le range dans la case libérée du côté gauche (ce qui libère une case du côté droit…),
- on cherche ensuite, de gauche à droite (à l'aide d'un indice `gauche`), un élément plus grand que le pivot ; lorsqu'on le trouve, on le range dans la case libérée du côté droit (ce qui libère une case du côté gauche…),
- on répète alternativement ces deux phases, jusqu'à ce que les deux indices (`droite` et `gauche`) se rejoignent ; le partitionnement est alors terminé, on peut ranger le pivot à sa place…

On obtient ainsi l'algorithme suivant :

```
Action Partitionner (     ES tab : Ttableau ; E deb, fin : entiers ;
                          S : posPivot : entier )
variables
       pivot, gauche, droite : entiers
       cherchePetit, trouvé : booléen
début
       # on sauvegarde le Pivot
  pivot ← tab[deb]
       # initialisations
  gauche ← deb ;  droite ← fin ;  cherchePetit ← vrai
       # partitionnement
  tantque ( droite > gauche )
  faire
       trouvé ← faux
       si ( cherchePetit )
       alors     # on cherche un "petit" du côté droit
            tantque ( (droite > gauche) et (non trouvé) )
            faire     si tab[droite] < pivot
                      alors trouvé ← vrai
                      sinon droite ← droite - 1
                      fin_si
            fin_tantque
                 # on le range du côté gauche
            tab[gauche] ← tab[droite] ;
                 # on diminue la partie gauche
            gauche ← gauche + 1
       Sinon     # on cherche un "grand" du côté gauche
            tantque ( (droite > gauche) et (non trouvé) )
            faire     si tab[gauche] > pivot
                      alors trouvé ← Vrai
                      sinon gauche ← gauche + 1
                      fin_si
            fin_tantque
                 # on le range du côté droit
            tab[droite] ← tab[gauche]
                 # on diminue la partie droite
            droite ← droite - 1
       fin_si
            # on change de sens
       cherchePetit ← non cherchePetit
       fin
       # on range le pivot à sa place
  tab[gauche] ← pivot
       # on range dans posPivot la position du pivot
  posPivot ← gauche
fin
```

On peut montrer que la complexité de cet algorithme est en moyenne en $\Theta(n \log n)$, donc optimale, mais avec une complexité dans le pire des cas en $\Theta(n^2)$.

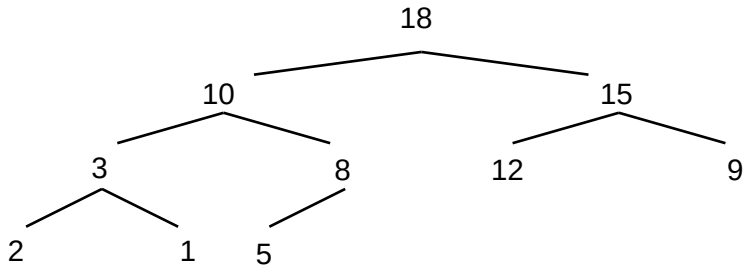
Chapitre 4. Le tri par tas : Heapsort

L'idée de base est la suivante : nous allons réaliser un tri par sélection (en cherchant d'abord l'élément maximal, puis l'élément maximal parmi les restants et ainsi de suite) mais en tirant profit des comparaisons déjà effectuées afin d'éviter de comparer inutilement des éléments déjà comparés…

Pour cela, nous allons utiliser une structure particulière appelée *tas* (en anglais *heap*) : un tas est un arbre binaire quasi-parfait ordonné de façon telle que tout sommet a une valeur supérieure ou égale aux valeurs de ses fils, et donc aux valeurs des éléments de ses sous-arbres.

Rappelons qu'un arbre binaire est quasi-parfait lorsque tous les « niveaux » sont remplis, à l'exception éventuelle du dernier niveau, auquel cas les éléments présents sont regroupés à gauche.

Voici par exemple un tas, dont le dernier niveau est incomplet :



Dans une telle structure, il est évidemment aisé de retrouver l'élément maximal : ce ne peut être que la racine !...

Un arbre binaire quasi-parfait peut être avantageusement représenté à l'aide d'un tableau de valeurs : les valeurs sont stockées de gauche à droite, niveau par niveau. Le tas précédent est ainsi représenté par le tableau suivant :

0	1	2	3	4	5	6	7	8	9
18	10	15	3	8	12	9	2	1	5

Cette représentation possède les propriétés suivantes (il est aisé de s'en convaincre) :

- la racine est en position `0`,
- le père d'un sommet en position `i` $\neq$ `0` est en position `i div 2`,
- le fils gauche, s'il existe, d'un sommet en position `i` est en position `2*i + 1`,
- le fils droit, s'il existe, d'un sommet en position `i` est en position `2*(i + 1)`.

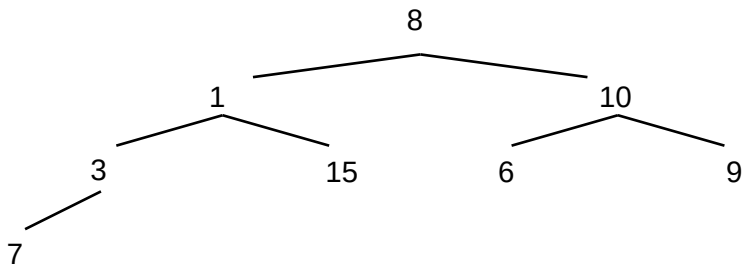
Le principe de l'algorithme de tri par tas va ainsi être le suivant :

- considérer le tableau à trier comme étant la représentation d'un arbre binaire quasi-parfait, et le « réorganiser » afin d'obtenir une structure de tas,
- supprimer la racine et réorganiser sous forme de tas la partie restante,
- répéter l'opération précédente jusqu'à épuisement des valeurs.

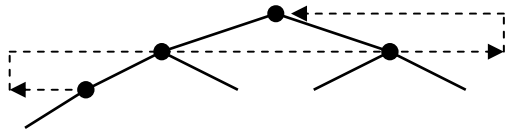
Voyons tout d'abord l'algorithme de réorganisation. Considérons par exemple le tableau suivant :

0	1	2	3	4	5	6	7
8	1	10	3	15	6	9	7

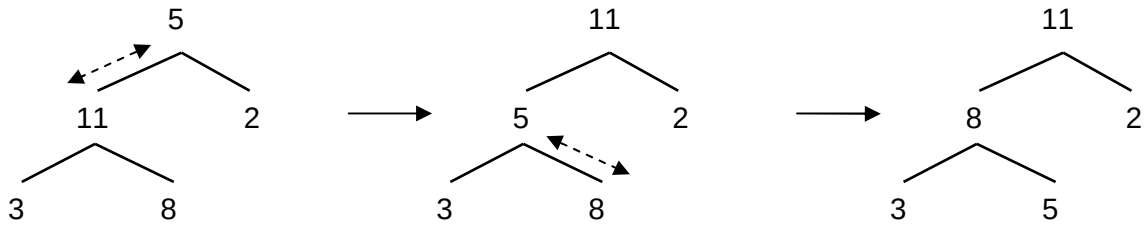
Ce tableau correspond à l'arbre binaire quasi-parfait ci-dessous :



Cet arbre n'est visiblement pas un tas. Nous allons le réorganiser, en procédant sous-arbre par sous-arbre, et ce de bas en haut et de droite à gauche (les sous-arbres vides ou réduits à un sommet sont évidemment déjà bien organisés) :



Réorganiser un sous-arbre consiste à vérifier que la racine est bien supérieure à chacun de ses fils. Dans le cas contraire, on fait remonter le plus grand fils à la place de la racine et on réitère ce processus jusqu'à ce que cette valeur ait trouvé sa place. Par exemple :



Grâce à l'ordre de parcours des sous-arbres, lorsqu'on traite un sous-arbre particulier seule la racine est éventuellement « mal placée ».

L'action `Réorganiser` peut alors s'écrire :

```
Action Réorganiser ( ES arbre : TTableau , E nbElem : entier )
# cette action réorganise arbre sous forme de tas
variables   racSousArbre : entiers
début
       # on parcourt les sous-arbres de droite à gauche et de bas en haut
  pour racSousArbre de (nbElem – 1) div 2 à 0 par pas de -1
  faire RangeRacine ( arbre, nbElem, racSousArbre
  fin_pour
fin
```

```
Action RangeRacine ( ES arbre : TTableau , E nbElem, rac : entier )
# cette action range la racine à sa place, les sous-arbres étant
# déjà organisés en tas
variables   posRacine, posGrandFils : entiers
début
  posRacine ← rac
```

```
        # tant que la valeur de la racine n'est pas en place,
        # on l'échange avec celle du plus grand fils…
   tantque ( non BienRangé (arbre, nbElem, posRacine) )
   faire
        posGrandFils ← NumPlusGrandFils (arbre, nbElem, posRacine)
        Echanger (arbre, pos, posGrandFils)
        posRacine ← posGrandFils
   fin_tantque
fin
```

```
Fonction Bien_Rangé ( arbre : TTableau ; nbElem, pos : entiers ) : booléen
# détermine si la valeur du sommet en position pos est "bien rangée"
# dans le tas par rapport à ses fils
début
   si ( 2 * pos ≥ nbElem )
   alors     # pos est feuille de l'arbre, donc ok
        retourner ( Vrai )
   sinon     # pos a au moins un fils
        retourner ( arbre[pos] ≥ arbre[numPlusGrandFils(arbre,nbElem,pos)] )
   fin_si
fin
```

```
Fonction NumPlusGrandFils
                (arbre : TTableau ; nbElem, pos : entiers) : entier
# détermine l'indice du fils de pos ayant la plus grand valeur
# (on sait que pos possède au moins un fils)
début
   si ( 2 * (pos + 1) = nbElem )
   alors     # pos n'a qu'un fils
        retourner (2*pos + 1)
   sinon     # on compare les valeurs des deux fils
        si arbre[2*pos + 1] > arbre[2*pos + 2]
        alors retourner (2*pos + 1)
        sinon retourner (2*pos + 2)
        fin_si
   fin_si
fin
```

```
Action Echanger (arbre : TTableau ; i, j : entiers)
# cette action échange dans arbre les valeurs en position i et j
variable    temp : entier
début
   temp ← arbre[i]
   arbre[i] ← arbre[j]
   arbre[j] ← temp
fin
```

Lorsque l'arbre est sous forme de tas, l'élément maximal se trouve à la racine. Il suffit ensuite de la supprimer, de réorganiser le tas, pour récupérer à la racine l'élément maximal suivant…

Voyons donc de quelle façon procéder. L'arbre binaire doit rester quasi-parfait. La seule valeur pouvant remplacer la racine est donc celle de la dernière feuille (la plus à droite du dernier niveau), en position `nbElem – 1`. Si nous mettons cette valeur en racine, il est alors nécessaire de réorganiser l'arbre, mais

seule la racine est éventuellement mal placée. Nous avons donc déjà tous les outils permettant de réaliser cette opération…

Finalement, l'algorithme de tri peut ainsi s'écrire :

```
Action Tri_Par_Tas ( ES tab : TTableau, nbElem : entier )
# cette action réalise un tri par tas du tableau tab
variables   sauveNb, S : entiers
début
      # on copie nbElem qui ne doit pas être modifié
  sauveNb ← nbElem
      # on réorganise le tableau en tas
  Réorganiser ( tab, sauveNb )
      # tant qu'il reste au moins 2 éléments, on échange la racine et
      # la dernière feuille, on diminue la taille du tas et
      # on range à sa place la nouvelle racine
  tant que ( sauveNb > 1 )
  faire
      Echanger (tab, sauveNb-1, 1)
      sauveNb ← sauveNb - 1
      RangeRacine ( tab, sauveNb, 1 )
  fin_tantque
  fin
```

Que peut-on dire de la complexité de cet algorithme ?

Remarquons tout d'abord qu'un tas à n éléments est de hauteur log n. Ainsi, l'action `RangeRacine` s'effectue en temps $\Theta(\log n)$, et donc l'action `Réorganiser` en temps $\Theta(n \log n)$. L'algorithme global s'effectue ainsi en temps $\Theta(n \log n)$, que ce soit au pire ou (on peut le montrer) en moyenne. Cet algorithme est donc de complexité optimale.

Chapitre 5. Les méthodes de tri externe (tri-fusion)

On parle de tri externe lorsqu'il s'agit de trier des données non stockées en mémoire centrale (il s'agit ainsi de tris de fichiers). Les méthodes utilisées dans ce cadre sont efficaces en termes de complexité en temps, mais utilisent de l'espace disque supplémentaire (généralement un espace de taille 2n pour trier n éléments).

Elles sont facilement convertibles en tris internes, en utilisant un tableau supplémentaire (en plus du tableau contenant les éléments à trier).

Nous présenterons ici le principe de ces algorithmes. Leur écriture (en version tri interne), constitue un excellent exercice !...

5.1. Tri balancé par monotonies de longueur 2^n

Une monotonie est une suite triée d'entiers. Le principe de cet algorithme est le suivant :

- on partage le fichier à trier F0 en deux fichiers F2 et F3 contenant des monotonies de longueur 1,
- on fusionne les monotonies des fichiers F2 et F3 et on répartit les monotonies obtenues (de longueur 2), dans les fichiers F0 et F1,
- on continue les fusions de monotonies, dont les longueurs seront de 4, 8, 16, etc., en alternant les paires de fichiers F0, F1 et F2, F3,
- on s'arrête dès que la longueur de la monotonie atteint ou dépasse le nombre d'éléments à trier.

Illustrons cette idée sur un exemple… Soit F0 le fichier contenant les entiers suivants :

F0 : 11 18 32 47 12 25 10 53 62 21

Etape 1 : on répartit (en alternant) les monotonies de longueur 1 de F0 sur F2, F3 :

F2 : 11 32 12 10 62

F3 : 18 47 25 53 21

Etape 2 : fusion des monotonies de longueur 1, réparties alternativement vers F0 et F1 (on obtient des monotonies de longueur 2)

F0 : 11 18 12 25 21 62

F1 : 32 47 10 53

Etape 3 : fusion des monotonies de longueur 2, réparties alternativement vers F2 et F3 (on obtient des monotonies de longueur 4)

F2 : 11 18 32 47 21 62

F3 : 10 12 25 53

Etape 4 : fusion des monotonies de longueur 4, réparties alternativement vers F0 et F1 (on obtient des monotonies de longueur 8)

F0 : 10 11 12 18 25 32 47 53

F1 : 21 62

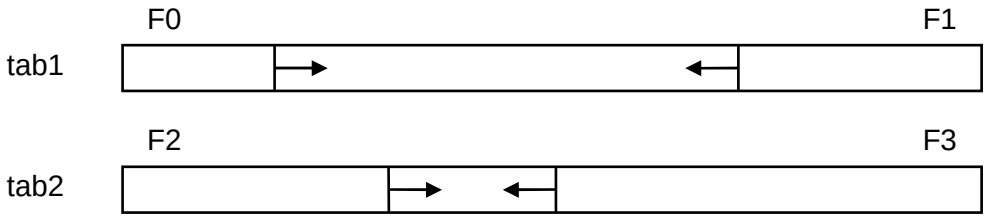
Etape 5 : fusion des monotonies de longueur 8, réparties alternativement vers F2 et F3 (on obtient monotonies de longueur 16)

F2 : 10 11 12 18 21 25 32 47 53 62

Le tri est terminé, le résultat est dans F2.

On notera qu'ici, le nombre de comparaisons et de transferts reste identique, quelle que soit la « configuration » des éléments à trier. La complexité est bien en $\Theta(n \log n)$, dans le meilleur et le pire des cas ainsi qu'en moyenne.

Pour implémenter cet algorithme en version « tri interne », on peut utiliser deux tableaux pour représenter les quatre fichiers (les fichiers F1 et F3 sont représentés « de droite à gauche » :



5.2. Tri balancé par monotonies naturelles

L'idée consiste à améliorer l'algorithme précédent en traitant les monotonies « comme elles se présentent », et non en considérant qu'elles sont de longueur prédéterminée. Ainsi, si le fichier est déjà trié, une seule étape suffira.

On peut facilement repérer la fin d'une monotonie, lorsqu'on rencontre un élément plus petit, ou lorsqu'on atteint la fin du fichier.

Sur l'exemple précédent, l'algorithme donne le résultat suivant :

Etape 1 : Eclatement de F0 sur F2, F3 (monotonies naturelles)

F0 : 11 18 32 47 12 25 10 53 62 21

donne F2 : 11 18 32 47 10 53 62

F3 : 12 25 21

Etape 2 : Fusion de F2, F3 sur F0, F1 (monotonies naturelles)

F0 : 11 12 18 25 32 47

F1 : 10 21 53 62

Etape 3 : Fusion de F0, F1 sur F2, F3 (monotonies naturelles)

F2 : 10 11 12 18 21 25 32 47 53 62

On a donc effectué deux étapes de moins.

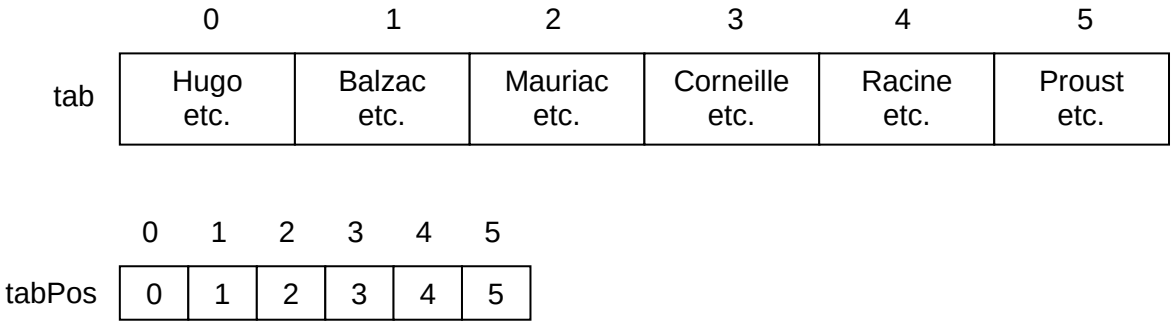
Dans le pire des cas, les monotonies sont au départ toutes de longueur 1 (le fichier est « trié inverse »). On retrouve dans ce cas exactement l'algorithme précédent. C'est le seul cas où l'on ne gagne rien...

La complexité de cet algorithme est toujours en $\Theta(n \log n)$ en moyenne et dans le pire des cas, mais cette fois en $\Theta(n)$, dans le meilleur des cas.

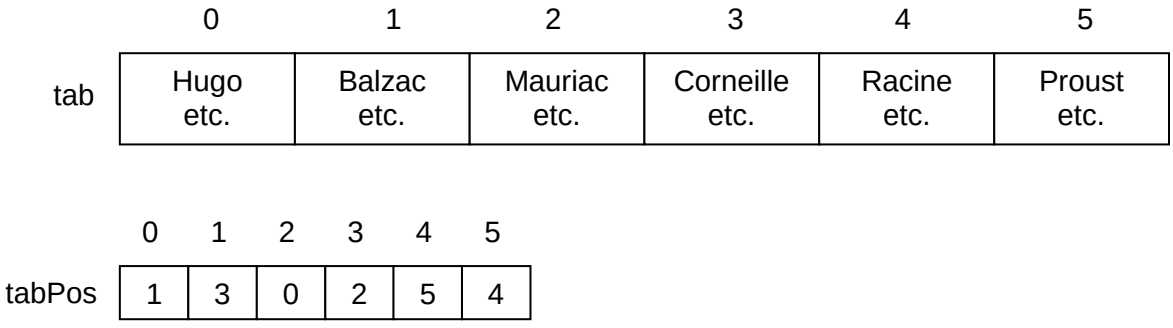
Chapitre 6. Tri de « grands objets »

Lorsque la taille des objets à trier est importante, il peut être coûteux d'effectuer des décalages ou des échanges. Dans ce cas, on utilisera de préférence un « tableau des positions » des éléments à trier et le tri s'effectuera sur ce tableau, ce qui permet d'éviter les déplacements d'objets (on se contentera de déplacer les positions des objets).

Illustrons ce principe sur un exemple : supposons que l'on souhaite trier un tableau `tab` d'informations concernant des étudiants, le tri devant s'effectuer sur le Nom des étudiants. On associe au tableau `tab` un tableau `tabPos`, initialisé de la façon suivante :



Initialement, le premier étudiant est en position 0, le second en position 1, etc. Le tableau n'est donc pas trié. Après le tri, nous aurons :



Le tableau `tab` n'a donc pas été modifié et la tableau `tabPos` indique que le premier étudiant est Balzac (en position 1 dans `tab`), le deuxième Corneille (en position 3 dans `tab`), etc.

Les algorithmes vus précédemment s'adaptent sans difficulté particulière à cette situation…