CONSERVATOIRE NATIONAL DES ARTS ET METIERS

PARIS

MEMOIRE

POUR L'EXAMEN PROBATOIRE

en

INFORMATIQUE

par

Nicolas HERVE

Les algorithmes de tri

Soutenu le 6 mai 2004

JURY

PRESIDENTE : Mme COSTA

Sommaire

Introduction

Classer les contacts de son carnet d'adresse par ordre alphabétique, trier les résultats d'une recherche sur internet par pertinence ou encore archiver ses mails les plus anciens sont des opérations courantes pour une personne qui utilise un ordinateur. Elles ont toutes en commun le fait de mettre en oeuvre une opération de tri sur des données. Mais ces quelques exemples anecdotiques, directement perceptibles par l'utilisateur, sont loin de représenter l'étendue du champ d'application du tri en informatique. Le tri des données est notamment présent dans de nombreux programmes en tant que phase intermédiaire. On retrouve ainsi des algorithmes de tri dans toutes les applications et jeux 3D pour l'affichage des facettes des objets qui sont triées selon leur éloignement (algorithme Z-sorting). Le tri est également massivement utilisé par les banques pour la gestion des opérations bancaires.

De nombreux théoriciens de l'informatique considèrent le tri comme le problème le plus fondamental en matière d'algorithmique. Le tri est une des principales opérations exécutées sur les ordinateurs. Des études tendent d'ailleurs à montrer qu'environ un quart des cycles machine sont utilisés pour trier[1].

De part l'ancienneté et l'importance de cette problématique, un grand nombre d'algorithmes et de variantes a été inventé. A travers cette profusion, comment choisir le bon algorithme ? La réponse à cette question n'est pas universelle et elle dépend de nombreux paramètres. Quelles sont les contraintes sur les données (leur type, leur nombre, leur organisation) ? Est-il important que l'algorithme n'utilise pas trop d'espace mémoire ? Quels sont les périphériques de stockage utilisés ? Nous allons aborder ces points et voir en quoi ils influent sur la problématique du tri des données.

Après un rappel sur les notions permettant l'analyse des algorithmes et de leurs performances, nous nous intéresserons aux principaux algorithmes en présentant une étude détaillée de leur comportement. Nous évoquerons également de manière plus brève d'autres algorithmes. La deuxième partie de ce mémoire sera consacrée à l'expérimentation avec la mise en pratique des algorithmes étudiés et la mesure de leurs performances en situation réelle d'une part et l'étude d'implémentations d'algorithmes de tri dans des outils professionnels d'autre part. Nous terminerons par une synthèse de ces résultats.

[1] Aggarwal et Vitter [6]

1. Rappels sur l'analyse des algorithmes

1.1 Qu'est-ce qu'un algorithme ?

On trouve de nombreuses définitions de ce qu'est un algorithme. Par exemple, dans le dictionnaire de l'Académie Française :
ALGORITHME n. m. XIII^{e} siècle, *augorisme.* Altération, sous l'influence du grec *arithmos,* «nombre», d'*algorisme,* qui, par l'espagnol, remonte à l'arabe *Al-Khuwarizmi,* surnom d'un mathématicien. MATH. Méthode de calcul qui indique la démarche à suivre pour résoudre une série de problèmes équivalents en appliquant dans un ordre précis une suite finie de règles.

Un algorithme est donc un ensemble d'opérations de calcul élémentaire, organisé selon des règles précises dans le but de résoudre un problème donné. Pour chaque donnée du problème, l'algorithme retourne une réponse correcte après un nombre fini d'opérations. On peut définir un problème comme un ensemble de contraintes qui existent sur des données fournies en entrée, ainsi que sur les données attendues en sortie et sur les relations entre elles. On parlera d'instance du problème pour une entrée particulière. On définira plus précisément ce qu'on appelle opération élémentaire par la suite.

On peut distinguer deux types d'algorithmes :

- les algorithmes séquentiels : les opérations élémentaires sont exécutées de manière séquentielle, c'est à dire l'une après l'autre, une seule à la fois.
- les algorithmes parallèles : des opérations élémentaires peuvent être exécutées en même temps (c'est le cas sur les ordinateurs à plusieurs processeurs ou pour certaines architectures de processeurs)

On ne s'intéressera ici qu'aux algorithmes séquentiels.

L'algorithmique (l'étude des algorithmes) a une double problématique :

- trouver au moins un algorithme pour répondre à un problème, et prouver qu'il fonctionne
- trouver un algorithme efficace

Les algorithmes de tri auxquels nous allons nous intéresser existent pour la plupart depuis plusieurs dizaines d'années. Ils ont tous fait l'objet de nombreuses études et on sait qu'ils fonctionnent correctement. Nous nous intéresserons donc plutôt à l'efficacité de ces algorithmes afin de pouvoir les comparer entre eux.

Une fois qu'un algorithme est choisi, il faut écrire le programme correspondant pour pouvoir le faire fonctionner. On dit qu'un programme est l'implémentation d'un algorithme. Cette implémentation se fait au moyen d'un langage de programmation particulier, pour un environnement particulier. Il peut donc exister plusieurs programmes pour un même algorithme. Les difficultés techniques (allocation mémoire, gestion des erreurs, gestion des entrées/sorties, ...) sont prises en compte au niveau du programme et non au niveau de l'algorithme. Les algorithmes sont ainsi concis et plus simples à étudier. Un algorithme est efficace pour toutes ses implémentations indépendamment de la programmation.

1.2 Analyse de l'efficacité d'un algorithme

L'étude de l'efficacité d'un algorithme porte sur deux principaux facteurs : le temps d'exécution et l'espace mémoire nécessaire pour résoudre un problème donné. Ces deux facteurs seront mesurés en fonction de la taille du problème fourni en entrée. Bien que les performances des ordinateurs ne cessent de croître de manière exponentielle, il est toujours important d'avoir des algorithmes performants, ne serait-ce que parce que la quantité de données que ces algorithmes doivent traiter est elle aussi en constante augmentation. Nous allons voir que de mauvais algorithmes peuvent rapidement devenir inutilisables sur des volumes de données conséquents. Enfin l'étude des algorithmes permet de mieux comprendre leur comportement et la manière dont ils réagissent en fonction des données, et donc d'avoir d'avantage d'informations permettant de choisir un algorithme pour une situation précise.

Afin d'analyser les algorithmes en faisant au maximum abstraction de leurs implémentations, il faut se doter d'un modèle pour les ressources systèmes et machines ainsi que leurs coûts. On utilisera un modèle générique basé sur une machine à accès aléatoire (RAM). Ce modèle contient les instructions classiques en informatique (opération élémentaire) :

- arithmétique : addition, soustraction, multiplication, division, modulo, partie entière, partie entière supérieure
- transfert de données : lecture, stockage, copie
- instructions de contrôle : branchement conditionnel et inconditionnel, appel de sous-routine et sortie de sous-routine

Il manipule des données du type entier et réel à virgule flottante.

Nous allons nous intéresser principalement aux performances en temps des différents algorithmes, on parle aussi de complexité en temps. Afin de pouvoir comparer les algorithmes nous allons, après une étude théorique de chacun d'entre eux, exprimer cette complexité temporelle en fonction de la taille du problème. Mais pour des problèmes de taille identique, on peut avoir pour un même algorithme des performances fondamentalement différentes en fonction d'autres paramètres sur les données (typiquement leur répartition pour les algorithmes de tri). Pour essayer de cerner au mieux ces comportements, on analyse la complexité des algorithmes dans le cas le plus favorable, le cas le plus défavorable et le cas moyen. Cette analyse théorique devant être au maximum indépendante de l'implémentation qui sera faite de l'algorithme, la complexité temporelle ne fournira pas un temps d'exécution en fonction de la taille du problème. Pour chaque algorithme il faut trouver les opérations élémentaires les plus significatives (les plus coûteuses en temps) et exprimer la complexité temporelle en fonction de ces grandeurs. Ensuite, si on souhaite avoir une estimation du temps d'exécution pour une implémentation précise, sur une machine précise, il ne reste qu'à obtenir le temps de traitement de chacune de ces opérations élémentaires. Dans cette étude des algorithmes de tri nous regarderons principalement le nombre de comparaisons et le nombre d'affectations nécessaires pour trier un ensemble de clés.

L'étude mathématique de la complexité d'un algorithme est souvent délicate. Or on n'a pas toujours besoin d'avoir une mesure exacte de cette complexité. Dès que la taille du problème devient suffisamment grande, seul l'ordre de grandeur de la complexité est important. On parle alors d'étude asymptotique. On utilise alors les notations mathématiques suivantes[2] :

[2] Sedgewick et Flajolet [7]

Etant donné une fonction $f(n)$,

- $O(f(n))$ est l'ensemble de tous les $g(n)$ tels que $|g(n)/f(n)|$ est borné supérieurement quand $n \to \infty$
- $\Omega(f(n))$ est l'ensemble de tous les $g(n)$ tels que $|g(n)/f(n)|$ est borné inférieurement par un nombre strictement positif quand $n \to \infty$
- $\Theta(f(n))$ est l'ensemble de tous les $g(n)$ tels que $|g(n)/f(n)|$ est borné inférieurement et supérieurement quand $n \to \infty$

La notation O exprime une borne supérieure, Ω une borne inférieure et Θ signifie que la borne supérieure concorde avec la borne inférieure.

Rappel :

O	: omicron (on dit également "grand O")
Θ	: thêta
Ω	: oméga
γ	: gamma
$\lg n$	: logarithme base 2 de n
$\ln n$	: logarithme népérien de n
$\log_b n$	: logarithme base b de n

On peut maintenant distinguer différents ordres de grandeur de complexité, qui vont définir différentes familles d'algorithmes :

1:	temps d'exécution constant
$\lg n$:	temps d'exécution logarithmique
n:	temps d'exécution linéaire
$n\lg n$:	temps d'exécution en $n\lg n$
n^2:	temps d'exécution quadratique
n^3:	temps d'exécution cubique
2^n:	temps d'exécution exponentiel

A titre d'illustration voici une comparaison des temps d'exécution pour les principales complexités que nous allons rencontrer (sur la base arbitraire d'une milliseconde pour une opération élémentaire):

$\lg n$	n	$n\lg n$	n^2
3 ms	10 ms	33 ms	100 ms
7 ms	100 ms	664 ms	10 s
10 ms	1 s	10 s	16 mn 40 s
13 ms	10 s	2 mn 13 s	1 j 3 h 46 mn
17 ms	1 mn 40 s	27 mn 41 s	115 j 17 h
20 ms	16 mn 40 s	5 h 32 mn	31 ans 259 j

On voit donc bien que même sur un ordinateur très puissant, la complexité de l'algorithme reste primordiale dans la détermination du temps d'exécution.

2. Les principaux algorithmes de tri

2.1 Le problème du tri

Nous nous intéresserons ici au problème du tri des données. On considère un ensemble de clés sur lequel une relation d'ordre totale est définie. On fournit à l'algorithme une suite de clés $\langle d_1, d_2, ..., d_n \rangle$ issues de cet ensemble. On doit alors obtenir en sortie une permutation de cette suite de sorte que $d'_1 \leq d'_2 \leq ... \leq d'_n$.

Rappel :
Une relation d'ordre est une relation binaire réflexive, antisymétrique et transitive. Un ensemble muni d'une relation d'ordre est un ensemble ordonné. $\leq$ est une relation d'ordre et $(E, \leq)$ est un ensemble ordonné si :

- *réflexive* : pour tout x dans E, $x \leq x$
- *antisymétrique* : pour tout x, y dans E, si $x \leq y$ et $y \leq x$ alors $x = y$
- *transitive* : pour tout x, y, z dans E, si $x \leq y$ et $y \leq z$ alors $x \leq z$

Si pour tout x, y dans E, on a soit $x \leq y$, soit $y \leq x$, alors la relation d'ordre est totale et E est totalement ordonné; sinon la relation d'ordre est partielle et E est partiellement ordonné.

Pour l'analyse des algorithmes de tri, nous prendrons naturellement comme taille du problème le nombre *n* de clés à trier.

Ce problème est relativement simple à appréhender et il existe de très nombreux algorithmes permettant de le résoudre. Ceci s'explique d'une part parce que ce problème est l'un des plus anciens de l'informatique, mais également parce qu'il n'existe pas d'algorithme universel, performant dans tous les cas. Nous allons voir que les hypothèses faites sur la nature des données à trier, ainsi que sur les contraintes matérielles (espace mémoire disponible, type de mémoire secondaire) influent sur les performances des algorithmes présentés.

R. Sedgewick explique que l'on a constamment besoin d'algorithmes de tri plus performants[3]. Selon la loi de Moore, la puissance des processeurs et la capacité mémoire des ordinateurs doublent tous les 18 mois. Mais la taille des problèmes suit l'évolution de la mémoire. Si un algorithme permet de trier *n* clés en un temps de $n\lg n$ sur un certain ordinateur, alors quel temps faut-il pour trier $2n$ clés sur un ordinateur 2 fois plus puissant ? La réponse est $(2n\lg 2n)/2 = n\lg n + n$. C'est à dire plus de temps !

L'objectif est ici de présenter une étude théorique de ces algorithmes et de leur complexité puis de valider cette étude par l'expérience. L'analyse asymptotique de la complexité donne un ordre de grandeur. Mais cette étude masque un certain nombre de facteurs constants qu'il peut être intéressant de connaître pour comparer plus finement des algorithmes dont la complexité asymptotique est similaire. Nous essaierons de déterminer ces facteurs constants par le déroulement de simulations sur ordinateur.

Une des principales contraintes concerne l'espace mémoire disponible. Est-on capable de trier l'ensemble des données en mémoire centrale? Le volume de données est-il plus important que la taille mémoire ? Dans ce cas il faudra avoir recours aux mémoires secondaires (disque dur ou bande magnétique). Cette contrainte est fortement structurante pour l'étude d'un algorithme

[3] Sedgewick [9]

de tri, et plus particulièrement pour l'étude de ses performances. On distingue donc deux types de tri :

- les tris internes : l'ensemble des données à trier peut être contenu en mémoire centrale. Dans ce cas, l'étude de la complexité temporelle de l'algorithme se basera sur les opérations élémentaires les plus coûteuses que sont les comparaisons entre deux données et éventuellement l'échange de deux données dans la suite à trier.
- les tris externes : l'ensemble des données à trier ne peut pas être contenu en mémoire centrale. Dans ce cas, les temps d'accès aux données sur la mémoire secondaire sont beaucoup plus coûteux que leur traitement en mémoire centrale. L'étude de la complexité temporelle se basera donc sur le nombre d'entrées / sorties nécessaires pour le tri.

Pour l'étude des algorithmes de tri, on s'intéressera aux tableaux d'entiers. Par convention, nous considérerons que l'indice d'un tableau de taille n va de 1 à n (et non de 0 à n-1 comme c'est le cas dans certains langages de programmation). Nous verrons dans la partie 4 l'influence du type des données et de leur distribution sur les performances des algorithmes.

Outre les performances, les algorithmes de tri ont d'autres caractéristiques qui les distinguent et qui peuvent être importantes :

- la stabilité : un algorithme stable apporte la garantie de ne pas modifier l'ordre initial des clés identiques
- tri sur place : l'algorithme ne nécessite pas (ou peu) de mémoire supplémentaire pour trier, il réarrange directement les clés dans le tableau fourni en paramètre

2.2 Les tris par comparaison

La plupart des algorithmes de tri que nous allons étudier est basé sur la comparaison deux à deux des clés qui doivent être ordonnées. Or on peut trouver un minorant du nombre de comparaisons que doit effectuer ce type d'algorithmes de tri.

On représente par un arbre de décision (arbre binaire plein) le déroulement d'un tri de n clés, avec chaque nœud interne représentant la comparaison entre deux clés du tableau et chaque feuille représentant les permutations devant être effectuées pour avoir le tableau trié[4]. Ne connaissant pas à l'avance l'ordre des clés au départ, tout algorithme de tri doit pouvoir effectuer les $n!$ permutations potentielles. L'arbre de décision a donc $n!$ feuilles. Si on pose h la hauteur de l'arbre (la longueur du plus long chemin partant de la racine), alors on sait que le nombre maximum de feuilles est 2^h. Cette hauteur h est le nombre maximum de comparaisons effectuées pour trier le tableau (cas défavorable). On a donc :

$$\begin{aligned} & n! \leq 2^h \\ \Leftrightarrow \quad & \lg(n!) \leq \lg(2^h) \\ \Leftrightarrow \quad & h = \Omega(n \lg n) \qquad (\text{car } \lg(n!) = \Theta(n \lg n)) \end{aligned}$$

Tout algorithme de tri par comparaison exige $\Omega(n \lg n)$ comparaisons dans le cas le plus défavorable.

[4] Voir [1, pages 160 et 161] pour plus de détails

2.3 Tri par insertion

Nom anglais : insertion sort - Propriétés : tri interne, sur place, stable

Cet algorithme de tri est un des plus simples qui existent. Son implémentation est facile à réaliser. Il n'est toutefois pas performant dès que le nombre de clés à trier devient conséquent. Son étude est donc une bonne introduction à l'étude des algorithmes de tri, même s'il a peu de chances d'être utilisé en dehors d'un but pédagogique.

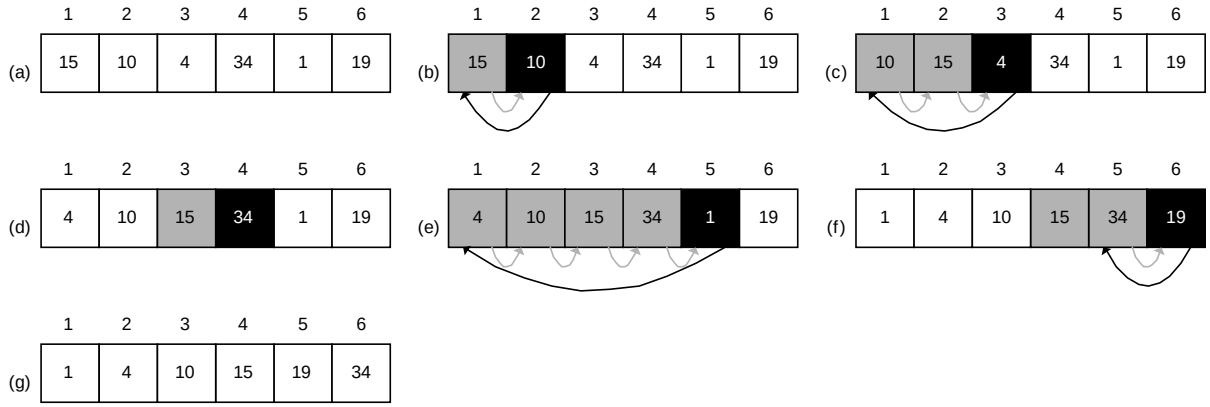
L'algorithme du tri par insertion est souvent assimilé au joueur qui trie ses cartes. Au début, le joueur a ses cartes placées en tas devant lui. Il les prend une par une de sa main droite pour les placer dans sa main gauche au bon endroit (c'est à dire en insérant systématiquement la dernière carte saisie à sa place).

La procédure suivante prend en paramètre un tableau A [1..*n*] contenant une suite de *n* clés à trier. Les clés étant triées sur place, ce même tableau A contient les clés triées à la sortie de la procédure.

Algorithme issu de [1, page 15]

```
TRI-INSERTION(A)
1       pour j ← 2 à n faire
2               clé ← A[j]
3               i ← j - 1
4               tant que i > 0 et A[i] > clé faire
5                       A[i + 1] ← A[i]
6                       i ← i - 1
7               fin tant que
8               A[i + 1] ← clé
9       fin pour
```

Exemple d'application sur la suite d'entiers ⟨15, 10, 4, 34, 1, 19⟩ :



Sur ce schéma on représente en noir la case qui est analysée (indice j de la boucle *pour*, ligne 1). Les cases grises correspondent aux valeurs comparées à A[*j*] (condition de la boucle *tant que*, ligne 4). Les flèches grises représentent les clés déplacées (ligne 5) et les flèches noires l'insertion de A[*j*] à sa place (ligne 8).

Etude de la complexité :

Cas favorable :
Le cas favorable se présente quand le tableau est déjà trié. Ainsi on n'entre pas dans la boucle *tant que* (lignes 4 à 7). On a alors une seule comparaison pour chaque passage dans la boucle *pour* principale. Le nombre total de comparaison est :

$$\sum_{j=2}^{n} 1 = n-1$$

De la même manière, on a 2 affectations par passage dans la boucle *pour*. Le nombre total d'affectations est :

$$\sum_{j=2}^{n} 2 = 2(n-1)$$

La complexité temporelle du tri par insertion dans le cas favorable est $\Theta(n)$.

Cas défavorable :
A l'opposé du cas favorable, le cas défavorable arrive lorsque le tableau est trié à l'envers. Dans ce cas, chaque nouvelle clé examinée dans la boucle *pour* principale doit être ramenée à l'indice 1 du tableau et il faut décaler l'ensemble des clés précédemment triées. Le nombre total de comparaisons est alors :

$$\sum_{j=2}^{n} \sum_{i=j-1}^{1} 1 = \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Le nombre total d'affectations est quant-à lui :

$$\sum_{j=2}^{n} \left(2 + \sum_{i=j-1}^{1} 1 \right) = \sum_{j=2}^{n} 2 + \sum_{j=2}^{n} (j-1) = (n-1)\left(2+\frac{n}{2}\right)$$

La complexité temporelle du tri par insertion dans le cas défavorable est $\Theta(n^2)$.

Cas moyen :
On considère que les clés sont réparties de manière uniforme. Pour un indice donné j de la boucle *pour*, la place de la clé située en A[j] peut être A[1], A[2] ... A[j]. Ces placements sont équiprobables. Le nombre de fois que l'on parcourt la boucle *tant que* est alors :

$$\frac{1}{j}\cdot 1 + \frac{1}{j}\cdot 2 + \ldots + \frac{1}{j}\cdot j = \frac{1}{j}\sum_{i=1}^{j} j = \frac{j+1}{2}$$

On a alors le nombre total de comparaisons :

$$\sum_{j=2}^{n} \frac{j+1}{2} = \frac{1}{2}\left(\sum_{j=2}^{n} j + \sum_{j=2}^{n} 1 \right) = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1 + n - 1 \right) = n\left(\frac{n+3}{4}\right) - 1$$

Et le nombre total d'affectations :

$$\sum_{j=2}^{n} \left(\frac{j+1}{2}\right) + 2 = \sum_{j=2}^{n} \frac{j+1}{2} + \sum_{j=2}^{n} 2 = n\left(\frac{n+11}{4}\right) - 3$$

La complexité temporelle du tri par insertion dans le cas moyen est $\Theta(n^2)$.

2.4 Diviser pour régner

Nom anglais : divide and conquer

Les algorithmes utilisant la stratégie diviser pour régner sont décomposés en 3 phases :

- *diviser* : si le nombre de clés est trop important pour être traité, alors diviser l'ensemble en n sous-ensembles disjoints.
- *régner* : appliquer l'algorithme de manière récursive à chacun des sous-ensembles
- *combiner* : fusionner les solutions des sous-ensembles pour obtenir la solution finale

L'étude de la complexité de ce type d'algorithmes récursifs peut se ramener à une récurrence au sens mathématique. Soit un algorithme qui pour traiter un problème de taille n le divise en a sous-problèmes de taille n/b ($a \geq 1$, $b > 1$). Si le problème est suffisamment petit ($n = 1$) il est résolu en un temps constant. Le temps nécessaire pour diviser le problème et pour combiner les solutions des sous-problèmes sont fonction de n (respectivement $D(n)$ et $C(n)$). On a alors la récurrence suivante :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n=1 \\ aT(n/b)+D(n)+C(n) & \text{sinon} \end{cases}$$

Il existe plusieurs méthodes pour résoudre ces récurrences. Nous utiliserons la méthode générale.

Présentation de la méthode générale

La méthode générale permet de résoudre les récurrences de la forme

$$T(n) = aT(n/b)+f(n) \text{ avec } a\geq 1,\ b>1 \text{ et } n>0$$

Le terme n/b peut ne pas être entier. On peut toutefois démontrer que le comportement asymptotique de cette récurrence n'est pas modifié si on le remplace par $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Pour des raisons de lisibilité, cette notation concernant les parties entières sera omise.
Le théorème général nous indique alors que pour résoudre cette récurrence on doit étudier les 3 cas suivants :

1) si $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$ pour une certaine constante $\varepsilon>0$, alors $T(n) = \Theta\left(n^{\log_b a}\right)$

2) si $f(n) = \Theta\left(n^{\log_b a}\right)$, alors $T(n) = \Theta\left(n^{\log_b a}\right)$

3) si $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ pour une certaine constante $\varepsilon>0$, et si $af(n/b)\leq cf(n)$ pour une certaine constante $c<1$ et pour tout n suffisamment grand, alors $T(n) = \Theta\left(f(n)\right)$

De manière générale, tout algorithme récursif consomme de l'espace mémoire pour la pile d'appel. Cette taille supplémentaire est masquée dans l'algorithme. Elle peut devenir conséquente dès lors qu'un grand nombre de paramètres passés par valeur entre en jeu. Il est possible de supprimer la récurrence de ces algorithmes en utilisant explicitement une pile. Nous ne présenterons pas les versions de ces algorithmes, mais on peut se référer à [8, page 128] pour les trouver.

2.5 Tri rapide

Nom anglais : quick sort - Propriétés : tri interne, sur place, non stable

Le tri rapide fait partie des algorithmes de tri du type "diviser pour régner". Bien que sa complexité temporelle dans le pire des cas soit $\Theta(n^2)$, c'est un des algorithmes les plus utilisés et également celui qui présente certainement le plus grand nombre de variantes.

Le tri rapide choisit un élément particulier de la liste de clés, appelé pivot. Il construit ensuite deux sous-listes gauche et droite contenant respectivement les clés inférieures et supérieures au pivot. Ainsi pour trier un sous-tableau A[*p*..*r*] du tableau initial A[1..*n*] on retrouve les 3 phases suivantes :

- *diviser* : choisir le pivot d'indice *q* dans le tableau A[*p*..*r*]. Partitionner en 3 ce sous-tableau :
  - A[*p*..*q*-1] contient les clés inférieures à A[*q*],
  - A[*q*] le pivot
  - A[*q*+1..*r*] contient les clés supérieures à A[*q*].

  Les deux sous-tableaux gauche et droite peuvent éventuellement être vides.
- *régner* : les sous-tableaux A[*p*..*q*-1] et A[*q*+1..*r*] sont traités en appelant récursivement le tri rapide.
- *combiner* : cette phase est instantanée. Puisque les sous-tableaux sont triés sur place, aucun travail n'est nécessaire pour les combiner.

2.5.1 Version standard

Algorithme issu de [1, page 140]

```
PARTITION(A, p, r)
1       x ← A[r]
2       i ← p - 1
3       pour j ← p à r - 1 faire
4               si A[j] ≤ x alors
5                       i ← i + 1
6                       PERMUTER(A, i, j)
7               fin si
8       fin pour
9       PERMUTER(A, i + 1, r)
10      retourner i + 1
```

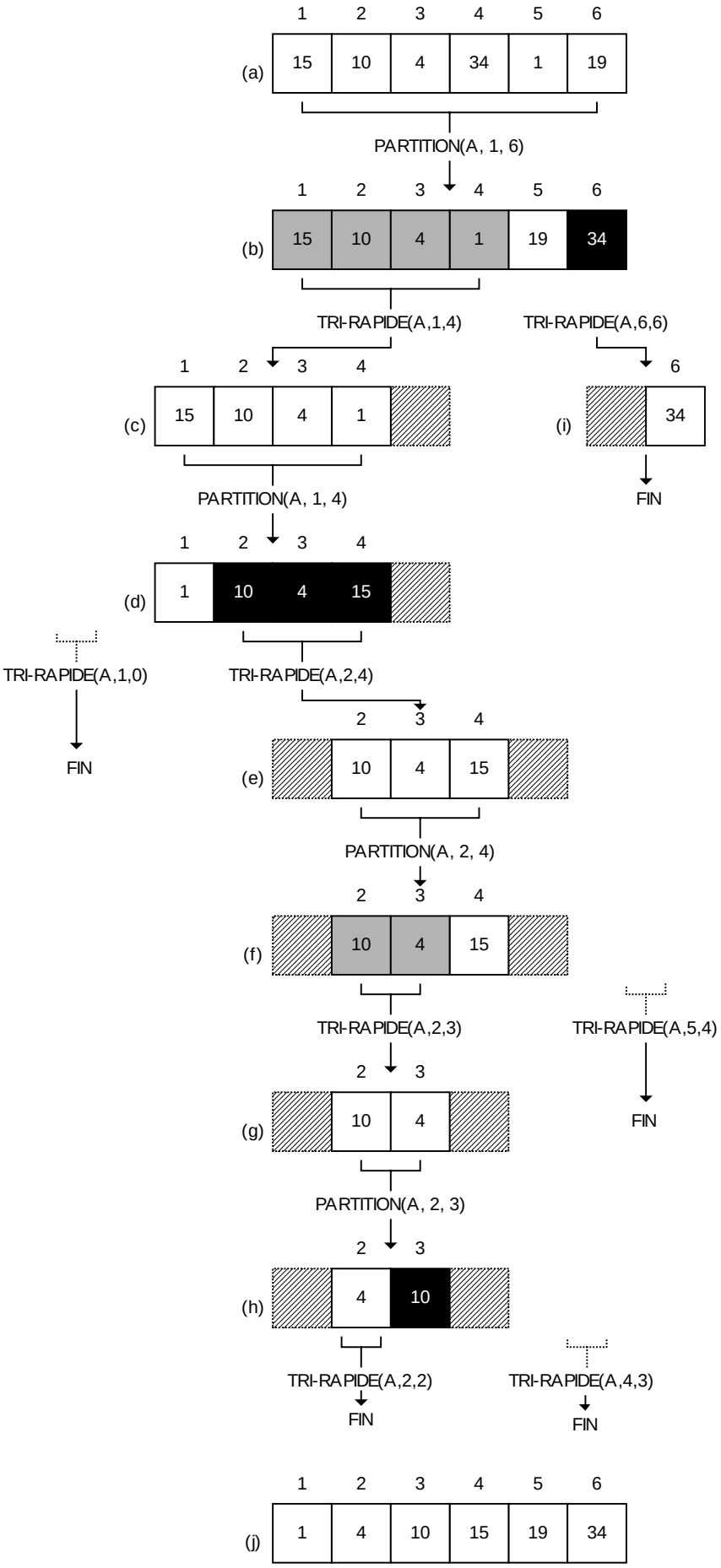
```
TRI-RAPIDE(A, p, r)
1       si p < r alors
2               q ← PARTITION(A, p, r)
3               TRI-RAPIDE(A, p, q - 1)
4               TRI-RAPIDE(A, q + 1, r)
5       fin si

PERMUTER(A, i, j)
1       tmp ← A[i]
2       A[i] ← A[j]
3       A[j] ← tmp
```

On démarre l'algorithme par un appel à TRI-RAPIDE(A, 1, *n*).

Cette version de l'algorithme prend la dernière clé du sous-ensemble comme pivot.

Exemple d'application sur la suite d'entiers ⟨15, 10, 4, 34, 1, 19⟩ :


1 2 3 4 5 6
(a) 15 10 4 34 1 19
PARTITION(A, 1, 6)
1 2 3 4 5 6
(b) 15 10 4 1 19 34
TRI-RAPIDE(A,1,4)
TRI-RAPIDE(A,6,6)
1 2 3 4
(c) 15 10 4 1
6
(i) 34
FIN
PARTITION(A, 1, 4)
1 2 3 4
(d) 1 10 4 15
TRI-RAPIDE(A,1,0)
TRI-RAPIDE(A,2,4)
FIN
2 3 4
(e) 10 4 15
PARTITION(A, 2, 4)
2 3 4
(f) 10 4 15
TRI-RAPIDE(A,2,3)
TRI-RAPIDE(A,5,4)
2 3
(g) 10 4
FIN
PARTITION(A, 2, 3)
2 3
(h) 4 10
TRI-RAPIDE(A,2,2)
TRI-RAPIDE(A,4,3)
FIN
FIN
1 2 3 4 5 6
(j) 1 4 10 15 19 34

Sur ce schéma on représente les appels récursifs aux deux procédures PARTITION et TRI-RAPIDE. Sur le tableau issu de la partition, on représente en blanc le pivot, en gris le sous-tableau gauche et en noir le sous-tableau droit. L'étape (j) est le tableau trié obtenu à la fin du déroulement de l'algorithme.

Etude de la complexité :
Les performances du tri rapide dépendent de la manière dont la procédure PARTITION parvient à créer deux sous-tableaux équilibrés ou non.

La complexité temporelle de la procédure PARTITION pour un tableau A[$p..r$] est $r-p+1$, donc pour un tableau A[1..n] on a $n-1$. On a déjà vu que la combinaison des solutions des sous-problèmes dans la récursion ne coûte rien. On se ramène donc au théorème général avec $f(n) = n-1$.

Cas favorable :
Le partitionnement fournit deux sous-tableaux de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil - 1$ à chaque appel récursif. Ce partitionnement équilibré nous donne la récurrence suivante :

$$T(n) \leq 2T(n/2)+\Theta(n) \text{ avec } a=2 \text{ et } b=2$$

On est dans le cas 2) du théorème général (puisque $n = n^{\log_2 2}$) donc

$$T(n) = \Theta(n\lg n)$$

La complexité temporelle du tri rapide dans le cas favorable est $\Theta(n\lg n)$.

Cas défavorable :
Le partitionnement fournit deux sous-tableaux de tailles 0 et $n-1$ à chaque appel récursif. Ce partitionnement est complètement déséquilibré, il donne la récurrence suivante :

$$T(n) = T(n-1)+n-1$$

$$\text{donc} \quad T(n) = \sum_{i=1}^{n} i-1 = \frac{n(n-1)}{2}$$

$$\text{donc} \quad T(n) = \Theta(n^2)$$

La complexité temporelle du tri rapide dans le cas défavorable est $\Theta(n^2)$.

Pour simplifier l'étude du cas moyen, nous allons introduire une première version améliorée du tri rapide.

2.5.2 Version aléatoire

Cette version rend le cas défavorable très improbable. En effet, au lieu de prendre la dernière clé du sous-tableau comme pivot, elle prend une clé au hasard. La probabilité d'être dans le cas défavorable (c'est à dire de choisir aléatoirement une clé qui à chaque étape produirait un partitionnement complètement déséquilibré) est très faible.

Algorithme issu de [1, page 148]

```
PARTITION-RANDOMISE(A, p, r)
1       i ← RANDOM(p, r)
2       PERMUTER(A, r, i)
3       retourner PARTITION(A, p, r)
```

```
TRI-RAPIDE-RANDOMISE(A, p, r)
1       si p < r alors
2               q ← PARTITION-RANDOMISE(A, p, r)
3               TRI-RAPIDE-RANDOMISE(A, p, q - 1)
4               TRI-RAPIDE-RANDOMISE(A, q + 1, r)
5       fin si
```

Etude du cas moyen :
On considère que les clés sont uniques dans la suite de données.

Soit C_n la suite donnant le nombre de comparaisons nécessaires pour trier un tableau de n clés. On a déjà vu que le nombre de comparaisons nécessaires au partitionnement pour un tableau de taille n est n - 1. Avec q indice de la clé choisie comme pivot par PARTITION, on a donc :

$$C_n = n-1+C_{q-1}+C_{n-q}$$

L'indice q du pivot peut prendre les valeurs de 1 à n de manière équiprobable. On a donc :

$$C_n = n-1+\frac{1}{n}\sum_{i=1}^{n}(C_{i-1}+C_{n-i}) \qquad (1)$$

Or on a :

$$\sum_{i=1}^{n}C_{i-1} = \sum_{i=1}^{n}C_{n-i} = C_0+C_1+C_2+...+C_{n-1}$$

Donc (1) devient :

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

$$\Leftrightarrow \quad nC_n = n(n-1)+2\sum_{i=1}^{n}C_{i-1}$$

$$\Leftrightarrow \quad nC_n = n(n-1)+2\sum_{i=1}^{n-1}C_{i-1}+2C_{n-1} \qquad (2)$$

Pour n-1 on a alors :

$$(n-1)C_{n-1} = (n-1)(n-2)+2\sum_{i=1}^{n-1}C_{i-1}$$

$$\Leftrightarrow \quad 2\sum_{i=1}^{n-1}C_{i-1} = (n-1)C_{n-1}-(n-1)(n-2) \qquad (3)$$

En intégrant (3) dans (2) on obtient :

$$nC_n = n(n-1)+(n-1)C_{n-1}-(n-1)(n-2)+2C_{n-1}$$

$$\Leftrightarrow \quad nC_n = (n+1)C_{n-1}+2(n-1)$$

$$\Leftrightarrow \quad \frac{C_n}{n+1} = \frac{C_{n-1}}{n}+\frac{2(n-1)}{n(n+1)} \qquad (4)$$

En posant $F_n = \frac{C_n}{n+1}$ on obtient dans (4):

$$F_n = F_{n-1}+\frac{2(n-1)}{n(n+1)}$$

$$\Leftrightarrow \quad F_n = F_{n-1}+\frac{2}{n+1}-\frac{2}{n(n+1)}$$

$$\Leftrightarrow \quad F_n = F_0+F_1+F_2+2\sum_{i=3}^{n}\frac{1}{i+1}-2\sum_{i=3}^{n}\frac{1}{i(i+1)}$$

$$\Leftrightarrow \quad F_n = F_0+F_1+F_2+2\sum_{i=3}^{n}\frac{1}{i+1}-2\left(\sum_{i=3}^{n}\frac{1}{i}-\sum_{i=3}^{n}\frac{1}{i+1}\right)$$

$$\Leftrightarrow \quad F_n = F_0+F_1+F_2+4\sum_{i=3}^{n}\frac{1}{i+1}-2\sum_{i=3}^{n}\frac{1}{i}$$

$$\Leftrightarrow \quad F_n = F_0+F_1+F_2+4\sum_{i=4}^{n+1}\frac{1}{i}-2\sum_{i=3}^{n}\frac{1}{i}$$

Or on sait que :

$$H_n = \sum_{i=1}^{n}\frac{1}{i} = \ln n+\gamma+O(\frac{1}{n}) \text{ avec } \gamma\approx 0{,}57721$$

Donc on a :

$$F_n = F_0+F_1+F_2+4(H_{n+1}-H_3)-2(H_n-H_2)$$

$$\Leftrightarrow \quad F_n = F_0+F_1+F_2+2H_n-4H_3+2H_2+\frac{4}{n+1}$$

$$\Leftrightarrow \quad F_n = F_0+F_1+F_2+2H_n-\frac{13}{3}+\frac{4}{n+1}$$

$$\Leftrightarrow \quad F_n = F_0+F_1+F_2-\frac{13}{3}+2(\ln n+\gamma)+O(\frac{1}{n}) \qquad (5)$$

En revenant à C_n on obtient :

$$\frac{C_n}{n+1} = \frac{C_0}{1}+\frac{C_1}{2}+\frac{C_2}{3}+2\ln n+2\gamma-\frac{13}{3}+O(\frac{1}{n})$$

Or on a $C_0 = 0$, $C_1 = 0$ et $C_2 = 1$

$$\Leftrightarrow \quad C_n = 2(n+1)\ln n+(n+1)(2\gamma-4)+O(1)$$

$$\Leftrightarrow \quad C_n \sim 2n\ln n-2{,}84558n$$

Or on a $\ln n = \frac{\lg n}{\lg e} \sim 0{,}69315\lg n$

$$\Leftrightarrow \quad \boxed{C_n \sim 1{,}38629n\lg n-2{,}84558n}$$

Par un calcul similaire, on trouve le nombre moyen d'échanges effectués :

$$S_n = (n+1)(H_{n+1}-1)-1+O(1)$$

$$\Leftrightarrow \quad \boxed{S_n \sim 0{,}69315n\lg n-0{,}42279n}$$

La complexité temporelle du tri rapide aléatoire dans le cas moyen est $\Theta(n\lg n)$.

2.5.3 Versions améliorées

Il existe de nombreuses versions "améliorées" du tri rapide. Elles diffèrent en fonction du point faible de l'algorithme standard qu'elles souhaitent corriger.

Bien que les performances asymptotiques du tri rapide soient largement meilleures que celles du tri par insertion, il n'en est pas de même sur les petits ensembles[5]. L'ordre de grandeur asymptotique ne fait pas apparaître un certain nombre de constantes qui sont négligeables devant *n* lorsque *n* est grand mais qui deviennent importantes pour *n* petit. Or un algorithme

[5] Voir chapitre 5.4

récursif comme le tri rapide traite beaucoup de petits ensembles puisqu'il partitionne le problème jusqu'à obtenir des ensembles de cardinalité 1. Une optimisation possible est de déléguer à un tri par insertion les ensembles dont la cardinalité est inférieure à une constante M. On peut trouver la valeur optimale de M en étudiant la récurrence suivante sur le nombre de comparaisons dans le cas moyen :

$$C_n = \begin{cases} n-1+\frac{2}{n}\sum_{i=1}^{n} C_{i-1} & \text{si } n > M \\ n\left(\frac{n+3}{4}\right)-1 & \text{sinon} \end{cases}$$

On étudie alors :

$$C_n = n-1+\frac{2}{n}\sum_{i=M+1}^{n}\left(C_{i-1}\right)+\frac{2}{n}\sum_{i=1}^{M}\left(\frac{i(i+3)}{4}-1\right)$$

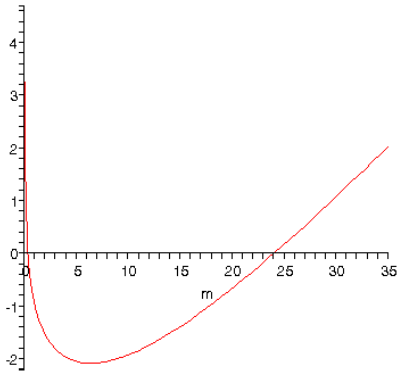
Par une démarche similaire à celle employée précédemment, on obtient pour $n > M$:

$$F_n = F_M+4\left(H_{n+1}-H_{M+1}\right)-2\left(H_n-H_M\right)$$

[...]

$$C_n \sim 2n\ln n+f(M)n \text{ avec } f(M) = \frac{\frac{M(M+3)}{4}-3}{M+1}-2\ln M$$

Voici le graphe de la fonction f :



Le minimum est atteint pour M ~ 6,355 (seule racine de la dérivée). On prendra M = 7 pour implémenter cette solution optimisée. Une autre option est de ne pas trier les tableaux dont la taille est inférieure à M. Lorsque le tri rapide se termine, il ne reste plus qu'à exécuter un tri par insertion sur l'ensemble du tableau. Globalement il faut voir que le tri rapide rapproche beaucoup plus vite que le tri par insertion les clés de leur emplacement final.

On a vu que le problème principal du tri rapide est de réussir à partitionner correctement (de manière équilibrée) l'ensemble à trier. La version aléatoire ne permet pas d'optimiser ce problème, elle évite juste le cas de la dégénérescence quadratique sur les tableaux déjà triés. Un moyen d'optimiser le tri rapide est donc de faire en sorte de mieux choisir le pivot. Au lieu de choisir une clé (choisie au hasard ou non), on va en choisir plusieurs que l'on va analyser pour savoir laquelle est la meilleure. Une possibilité est de choisir 3 clés et de prendre comme pivot celle qui a la valeur médiane des 3. On appelle alors cette variante le tri rapide médiane-de-trois. On peut éventuellement choisir parmi plus de 3 éléments. Une autre variante, dite pseudo-médiane-de-neuf consiste à choisir comme pivot la clé médiane de 3 valeurs médiane de 3 ensembles de 3 clés (médiane de 3 médianes). Comme pour la version incluant le tri par

insertion il est possible de mélanger les versions du tri rapide en fonction des cas pour avoir une version améliorée.

Il existe de nombreuses études sur l'amélioration du tri rapide. Nous en avons choisi 3.

● En 1993 J. Bentley et D. McIlroy implémentent une version de tri rapide améliorée pour la librairie standard C. Il s'agit d'un mélange du tri rapide standard, de la version médiane-de-trois et de la version pseudo-médiane-de-neuf. En 2003, M. Durand [10] montre que la complexité asymptotique (en considérant le nombre de comparaisons) est :

$$C_n = 1{,}5697n\ln n - 1{,}1512n + 1{,}5697\ln n - 7{,}4633 + o(1)$$

C'est à dire :

$$C_n \sim 1{,}088n\lg n - 1{,}1512n$$

Ceci est obtenu avec la règle suivante :

$n < 15$	tri rapide standard
$15 \leq n < 86$	tri rapide médiane-de-trois
$86 \leq n$	tri rapide pseudo-médiane-de-neuf

● S. Bhutoria et G. Konjevod [11] proposent une version qui profite de la phase de partitionnement pour choisir les deux pivots de l'étape suivante. Puisqu'un parcours des clés doit être effectué pour ce partitionnement, on essaye d'obtenir la valeur médiane de chacun des deux sous-ensembles qui vont être générés.

● J. Bentley et R. Sedgewick [9] proposent en 2002 une version de tri rapide aléatoire basée sur un partitionnement en 3 permettant de mieux gérer les cas d'égalité de clés. Ils annoncent que cette version est optimale et qu'aucun algorithme de tri basé sur des comparaisons ne peut mieux faire.

2.6 Tri fusion

Nom anglais : merge sort - Propriétés : tri interne et externe, stable, ne trie pas sur place

2.6.1 Version interne

Le tri fusion fait également partie des algorithmes de tri du type "diviser pour régner". Cet algorithme partage le tableau en deux sous-tableaux de taille $n/2$ qu'il trie. Il fusionne ensuite les résultats des deux sous-tableaux. On retrouve alors les 3 phases suivantes :

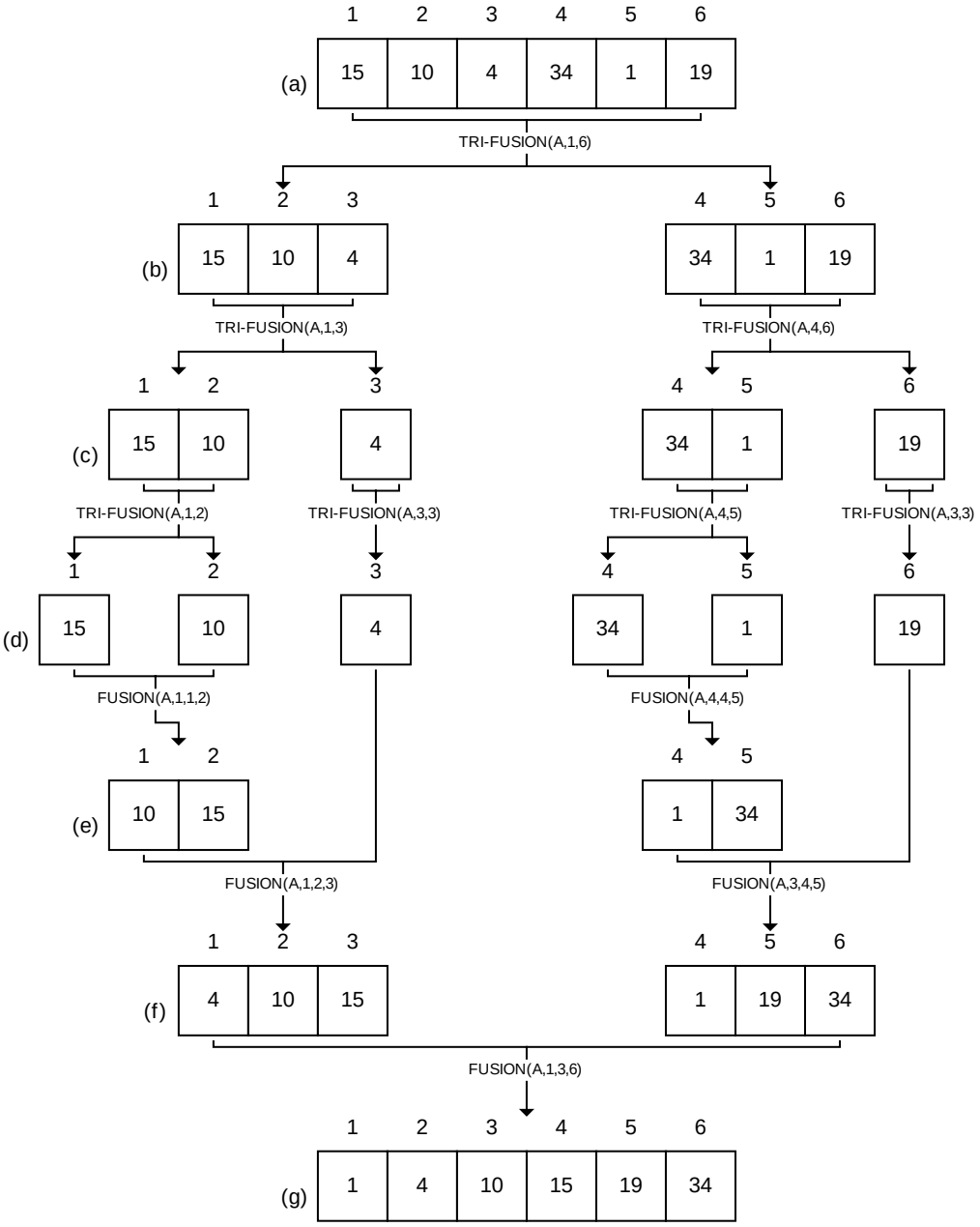
- *diviser* : partager le tableau en deux sous-tableaux de taille $n/2$. Cette phase est instantanée puisqu'il suffit de calculer l'indice $n/2$.
- *régner* : les sous-tableaux sont traités en appelant récursivement le tri fusion.
- *combiner* : c'est cette phase qui contient toute la logique de l'algorithme. La fusion de deux sous-tableaux déjà triés se fait en les parcourant en parallèle et en plaçant systématiquement la plus petite clé dans le tableau résultat. Pour cela la procédure FUSION crée deux tableaux temporaires pour stocker les deux sous-tableaux à fusionner. On utilise une sentinelle à la fin de chacun de ces 2 tableaux temporaires pour éviter d'ajouter des tests supplémentaires pour détecter la fin de l'un d'entre eux dans la procédure de fusion.

Algorithme issu de [1, page 27]

```
FUSION(A, p, q, r)
1     n1 ← q - p + 1
2     n2 ← r - q
3     créer tableaux L[1..n1 + 1] et R[1..n2 + 1]
4     pour i ← 1 à n1 faire
5           L[i] ← A[p + i - 1]
6     fin pour
7     pour j ← 1 à n2 faire
8           R[j] ← A[q + j]
9     fin pour
10    L[n1 + 1] ← ∞
11    R[n2 + 1] ← ∞
12    i ← 1
13    j ← 1
14    pour k ← p à r faire
15          si L[i] ≤ R[j] alors
16                A[k] ← L[i]
17                i ← i + 1
18          sinon
19                A[k] ← R[j]
20                j ← j + 1
21          fin si
22    fin pour
```

```
TRI-FUSION(A, p, r)
1     si p < r alors
2           q ← (p + r) / 2
3           TRI-FUSION(A, p, q)
4           TRI-FUSION(A, q + 1, r)
5           FUSION(A, p, q, r)
6     fin si
```

Exemple d'application sur la suite d'entiers ⟨15, 10, 4, 34, 1, 19⟩ :



Sur ce schéma, on représente le déroulement du tri par fusion. Les étapes *b*, *c* et *d* correspondent aux appels récursifs à la procédure TRI-FUSION. Ensuite les étapes *e*, *f* et *g* correspondent à la terminaison de la récursion par l'appel à la procédure FUSION.

Etude de la complexité :
Le tri par fusion est insensible aux données qu'il trie. En effet si on regarde de plus près l'algorithme, on se rend compte que la boucle principale (lignes 14 à 22) effectue systématiquement le même nombre d'opérations, quel que soit l'ordre relatif des clés des tableaux L et R. On en déduit donc qu'il n'y a pas de cas favorable ou défavorable. Toutes les entrées de taille *n* seront traitées avec un temps identique.
La procédure FUSION appliquée à un tableau de taille *n* effectue $2n + 2$ affectations et *n* comparaisons. Elle est donc en $\Theta(n)$. On a vu que la partition du problème dans TRI-FUSION est en $\Theta(1)$.

On se ramène donc à la récurrence suivante :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n=1 \\ 2T(n/2)+\Theta(n) & \text{sinon} \end{cases}$$

On a déjà vu dans l'étude du cas favorable du tri rapide que cette récurrence, d'après le théorème général, nous donne le résultat suivant :

$$T(n) = \Theta(n\lg n)$$

La complexité temporelle du tri fusion est donc $\Theta(n\lg n)$ dans tous les cas.

Contrairement au tri rapide, le tri fusion commence par partitionner le problème en deux sous-problèmes de tailles égales avant d'appliquer une procédure en $\Theta(n)$. On est donc toujours dans le cas du partitionnement favorable du tri rapide. Ceci nous assure que le nombre d'étapes de partitionnement sera minimum.

Toutefois, cet algorithme ne trie pas sur place. Il nécessite un espace mémoire supplémentaire en $\Theta(n)$. Pour s'en convaincre, il suffit d'envisager la dernière fusion qui doit être faite. Elle correspond à la partition du tableau initial. On a donc 2 sous-tableaux temporaires de taille $n/2$.

Comme pour le tri rapide, il est possible d'optimiser le tri fusion en lui substituant un tri par insertion pour les petits tableaux.

2.6.2 Version externe

Pour de très nombreuses applications, la quantité de données à trier est bien supérieure à la capacité mémoire de l'ordinateur. Il faut dans ce cas un algorithme particulier qui va s'appuyer sur les mémoires secondaires (disques durs ou bandes magnétiques) pour trier. La première contrainte est que les temps d'accès de ces mémoires secondaires est tellement grand, qu'il n'est pas comparable avec les temps d'accès et de traitement en mémoire centrale. La complexité de ces algorithmes est donc étudiée en fonction du nombre d'entrées/sorties plutôt que du nombre d'échanges ou de comparaisons. La seconde contrainte est lié au type et au nombre de mémoires secondaires. En effet, contrairement à un disque dur, une bande magnétique ne peut être accéder que séquentiellement.
Nous nous intéresserons ici à une version du tri par fusion utilisée avec des disques durs au sein des bases de données. Pour une version sur bandes magnétiques, se reporter à [8, chapitre 13] ou [2, chapitre 5].

On suppose qu'on a n clés à trier et qu'on dispose d'un espace en mémoire centrale permettant de stocker uniquement M clés.
La première étape est de créer des partitions de l'ensemble des clés qui puissent tenir en mémoire centrale. On lit donc les clés par blocs de taille M. Chaque bloc est trié en mémoire centrale à l'aide d'un des algorithmes vu précédemment. Il est ensuite réécrit sur le disque pour libérer l'espace mémoire pour trier les blocs suivants. A la fin de cette première étape, on a donc $\lceil n/M \rceil$ partitions triées de taille M. Le coût de cette phase est de n lectures et n écritures.
La seconde étape, comme pour le tri fusion classique, consiste à interclasser les partitions triées. Pour cela, on lit en parallèle les clés de M-1 partitions et on stocke la clé minimum dans le dernier espace mémoire afin de l'écrire sur le disque dans la nouvelle partition résultant de cette fusion. A la fin de cette étape, on a donc $\lceil n/M(M-1) \rceil$ partitions de taille M(M-1) chacune (sauf éventuellement la dernière). On recommence cette seconde étape

jusqu'à obtenir une partition unique qui sera le résultat final. Le coût de cette étape est également de *n* lectures et *n* écritures. Cette étape est répétée environ $\log_{M-1} n$ fois.

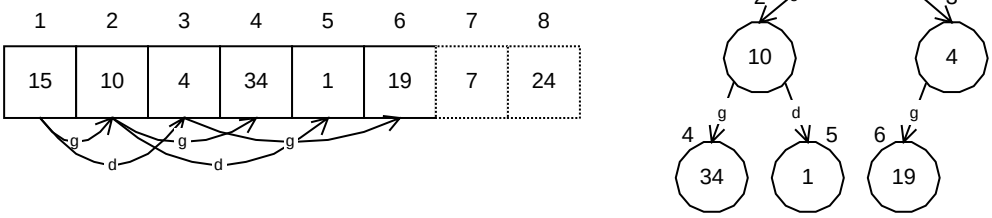
Le coût total en entrées/sorties est en $O(n \log_M n)$.

Les pistes d'optimisation pour ce tri sont principalement liées à l'utilisation en parallèle de plusieurs disques / bandes. On n'économise pas forcément en nombre d'entrées / sorties mais en temps.

2.7 Tri par tas

Nom anglais : heap sort - Propriétés : tri interne, non stable, sur place

Le tri par tas est un tri qui se base sur une structure de données particulière : le tas. Il s'agit d'une représentation d'un arbre binaire sous forme de tableau. L'arbre est presque complet : il est complètement rempli à tous les niveaux, sauf potentiellement le dernier. Le parcours de l'arbre (trouver le père ou les fils droit et gauche d'un nœud) se fait par un calcul d'indice sur le tableau. Pour un nœud d'indice *i*, on a le père à l'indice $\lfloor i/2 \rfloor$, le fils gauche à l'indice 2*i* et le fils droit à l'indice 2*i* + 1. On distingue la longueur du tableau, qui est le nombre d'éléments présents dans le tableau, de la taille du tableau, qui est le nombre d'éléments du tas. Il se peut que les derniers éléments du tableau ne fassent donc pas partie du tas. Cette petite distinction permet d'effectuer le tri sur place en conservant le tas et les clés déjà triées dans le même tableau et en les distinguant à l'aide de ces deux indices. Dans l'exemple suivant, la longueur du tableau est 8, alors que sa taille est 6. Ainsi les deux dernières cases ne font pas partie du tas :



Il existe deux types de tas : les tas max et les tas min. Ils correspondent à la manière dont les nœuds sont organisés. Dans un tas max, le père a toujours une valeur supérieure ou égale à celles de ses 2 fils. L'élément racine est donc l'élément maximum du tas. C'est le contraire dans un tas min. Le tri par tas se base donc sur un tas max pour trier le tableau sous-jacent.

Algorithme issu de [1, page 124]

```
PARENT(i)
1       retourner i/2

GAUCHE(i)
1       retourner 2i

DROITE(i)
1       retourner 2i + 1
```

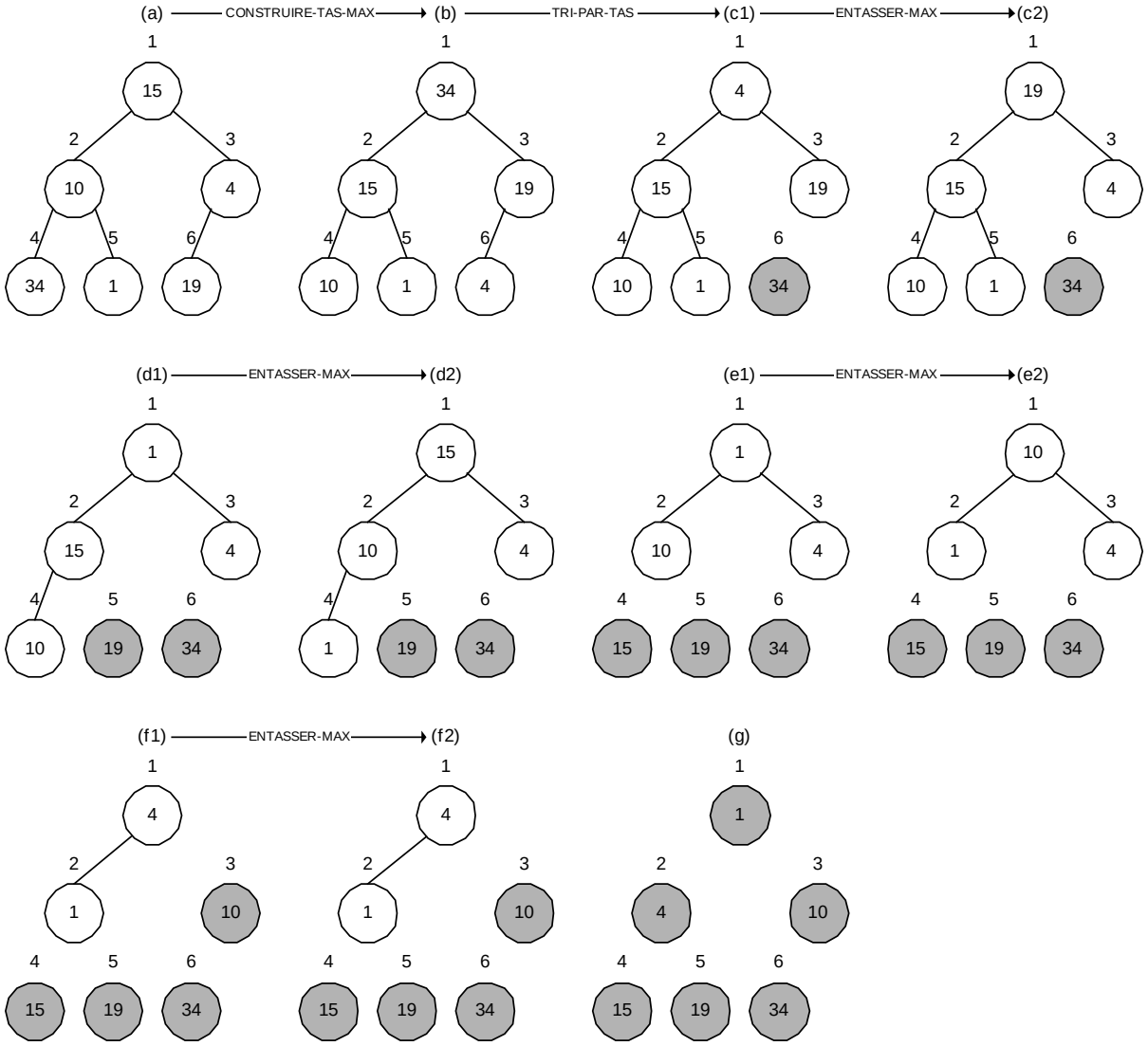
```
ENTASSER-MAX(A, i)
1       l ← GAUCHE(i)
2       r ← DROITE(i)
3       si l ≤ taille[A] et A[l] > A[i] alors
4               max ← l
5       sinon
6               max ← i
7       fin si
8       si r ≤ taille[A] et A[r] > A[max] alors
9               max ← r
10      fin si
11      si max ≠ i alors
12              PERMUTER(A, i, max)
13              ENTASSER-MAx(A, max)
14      fin si
```

```
CONSTRUIRE-TAS-MAX(A)
1       taille[A] ← longueur[A]
2       pour i ← longueur[A]/2 à 1 faire
3               ENTASSER-MAX(A, i)
4       fin pour
```

```
TRI-PAR-TAS(A)
1       CONSTRUIRE-TAS-MAX(A)
2       pour i ← longueur[A] à 2 faire
3               PERMUTER(A, 1, i)
4               taille[A] ← taille[A] - 1
5               ENTASSER-MAX(A,1)
6       fin pour
```

La procédure ENTASSER-MAX(A, *i*) est chargée de faire descendre la valeur du nœud *i* dans le tas afin de conserver la propriété du tas qui est qu'un père est toujours supérieur ou égal à ses fils. La procédure CONSTRUIRE-TAS-MAX(A) permet de réarranger un tableau pour le transformer en tas max. Enfin la procédure TRI-PAR-TAS(A) utilise les propriétés du tas pour trier un tableau. Elle commence par créer un tas max à partir du tableau. L'élément racine du tas est alors le maximum. C'est donc en théorie l'élément qui doit se trouver en dernière position du tableau trié. On échange donc cet élément avec le dernier du tableau, et on décrémente la taille du tas. Ainsi on a un tas qui va de 1 à *n* - 1 et en position *n* on a l'élément maximum du tableau qui est à sa place. L'élément qui se retrouve maintenant à la racine du tas n'est potentiellement plus le maximum. Un appel à ENTASSER-MAX sur cet élément va permettre de le replacer au bon endroit dans le tas. On a maintenant un tas max correct de taille *n* - 1. On peut recommencer jusqu'à épuisement du tas pour obtenir le tableau correctement trié.

Exemple d'application sur la suite d'entiers ⟨15, 10, 4, 34, 1, 19⟩ :



Sur ce schéma on représente en gris les éléments déjà triés. Pour chaque indice *i* de la boucle *pour* de la procédure TRI-PAR-TAS on a représenté les deux étapes du placement du maximum à sa position finale puis de l'appel à ENTASSER-MAX pour réorganiser le tas.

Etude de la complexité :

ENTASSER-MAX :
On étudiera, sans nuire à la généralité, uniquement les tas complets, c'est à dire les tas dont la taille vérifie :

$$\text{taille}[A] = \sum_{i=0}^{k} 2^i \qquad (k \geq 0)$$

La procédure ENTASSER-MAX appelée sur un tas de taille *n* utilise un temps constant pour comparer le père et ses deux fils. Il y a ensuite un éventuel appel récursif sur un des sous-arbres pour le cas où un échange a eu lieu. La taille de ce sous-arbre est *n*/2. On a alors la récurrence suivante :

$$T(n) \leq T(n/2) + \Theta(1)$$

En utilisant le cas 2) du théorème général avec a = 1 et b = 2 on obtient alors :

$T(n) \leq \Theta(\lg n)$

$\Leftrightarrow \quad T(n) = O(\lg n)$

On peut également borner le temps d'exécution de cette procédure sur un nœud de hauteur h par $O(h)$.

La complexité de la procédure ENTASSER-MAX est en $O(\lg n)$.

CONSTRUIRE-TAS-MAX :
On peut facilement voir que la procédure CONSTRUIRE-TAS-MAX appelle $n/2$ fois ENTASSER-MAX. Donc on peut borner sa complexité par $O(n \lg n)$.
Il est possible d'obtenir un majorant plus précis. On constate que la hauteur d'un tas est $\lg n$ et que le nombre d'éléments présents à la hauteur h est $n/2^{h+1}$. La complexité de la procédure est alors :

$$\sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^h} \right) \qquad (1)$$

Or on sait que :

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

Donc :

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Donc (1) devient :

$$O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^h} \right) \leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$\Leftrightarrow \quad O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^h} \right) = O(2n)$$

La complexité de la procédure CONSTRUIRE-TAS-MAX est en $O(2n)$.

TRI-PAR-TAS :
Cette procédure appelle une fois CONSTRUIRE-TAS-MAX et n - 1 fois ENTASSER-MAX. On peut donc borner sa complexité par $O(2n) + (n - 1) O(\lg n)$.
La complexité de la procédure TRI-PAR-TAS est en $O(n \lg n)$.

2.8 Tri par dénombrement

Propriétés : tri interne, stable, ne trie pas sur place

On a vu jusqu'à présent des algorithmes de tri par comparaison. On sait que pour les meilleurs d'entre eux la complexité est en $\Theta(n\lg n)$. En faisant quelques hypothèses sur les clés à trier, on peut utiliser des algorithmes ne faisant pas appel à la comparaison, et ainsi obtenir des complexités temporelles meilleures.

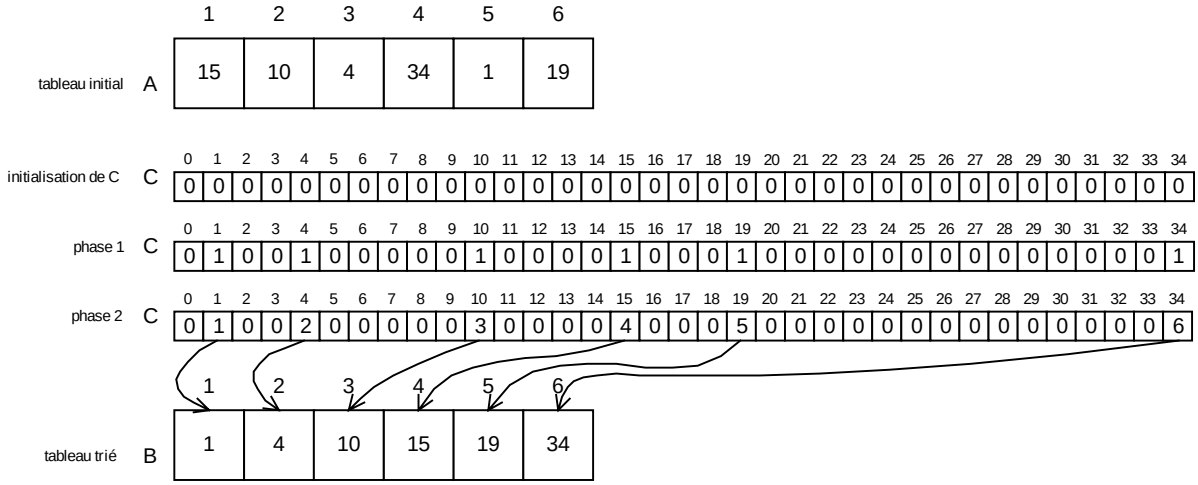
Ainsi le tri par dénombrement prend comme hypothèse que l'on trie des entiers et que l'on connaît l'intervalle 0..*k* dans lequel sont choisies les *n* clés à trier. Le tri par dénombrement détermine pour chaque clé le nombre de clés qui lui sont inférieures. Avec cette information, la clé peut être placée au bon endroit dans le tableau trié par un simple calcul d'adresse. Comment faire pour déterminer le nombre de clés inférieures à une clé donnée sans utiliser de comparaison? C'est là le rôle des hypothèses réductrices faites sur les clés. Comme on ne trie que des entiers et que les tableaux sont indicés par des entiers, la valeur d'une clé sert également d'indice de tableau.

Algorithme issu de [1, page 124]

```
TRI-DENOMBREMENT(A, B, k)
1       pour i ← 0 à k faire
2               C[i] ← 0
3       fin pour
4       pour j ← 1 à n faire
5               C[A[j]] ← C[A[j]] + 1
6       fin pour
7       pour i ← 1 à k faire
8               C[i] ← C[i] + C[i - 1]
9       fin pour
10      pour j ← n à 1 faire
11              B[C[A[j]]] ← A[j]
12              C[A[j]] ← C[A[j]] - 1
13      fin pour
```

Le tri a besoin de deux tableaux supplémentaires pour trier. Le tableau A[1..*n*] est le tableau initial contenant les clés à trier. Le tableau B[1..*n*] contient le résultat final. Le tableau C[0..*k*] sert à dénombrer le nombre de clés inférieures à une clé donnée.
La première phase (boucle des lignes 4 à 6) place dans chaque case C[*i*] le nombre de clés égales à *i* dans le tableau A. Ensuite, la seconde phase (boucle des lignes 7 à 9) fait la somme des valeurs obtenues afin d'avoir dans chaque case C[*i*] le nombre de clés inférieures ou égales à *i*. Il ne reste alors qu'à lire dans C l'indice final de chacun des éléments du tableau A pour les placer dans le tableau B (boucle des lignes 10 à 13).

Exemple d'application sur la suite d'entiers ⟨15, 10, 4, 34, 1, 19⟩ :



Etude de la complexité :

Cet algorithme est insensible à la distribution des données ainsi qu'à la présence ou non de clés en plusieurs exemplaires. On n'a donc pas de cas favorable ou défavorable.

La première boucle (initialisation de C) est en $\Theta(k)$.
La seconde boucle (phase 1) est en $\Theta(n)$.
La troisième boucle (phase 2) est en $\Theta(k)$.
La dernière boucle (remplissage de B) est en $\Theta(n)$.

Au total, la complexité de ce tri est donc $\Theta(2n+2k)$.

L'espace mémoire supplémentaire nécessaire pour effectuer le tri est en $\Theta(n+k)$.

3. D'autres algorithmes

3.1 Tri à bulles

Nom anglais : bubble sort - Propriétés : tri interne, non stable, sur place
Complexité dans tous les cas en $\Theta(n^2)$

Le principe de cet algorithme est d'échanger les clés contiguës qui ne sont pas correctement triées. Son nom vient du fait que les clés se déplacent comme des bulles dans une flûte de champagne. Elles remontent d'ailleurs si lentement que cela explique la complexité quadratique.

Algorithme issu de [1, page 35]

```
TRI-BULLES(A)
1       pour i ← 1 à n faire
2               pour j ← n à i + 1 faire
3                       si A[j] < A[j - 1] alors
4                               PERMUTER(A, j, j - 1)
5                       fin si
6               fin pour
7       fin pour
```

3.2 Tri par sélection

Nom anglais : selection sort - Propriétés : tri interne, stable, sur place
Complexité dans tous les cas en $\Theta(n^2)$

Le principe de cet algorithme est de choisir l'élément minimum et de le placer en première position, puis le second placé en seconde position, ...

```
TRI-SELECTION(A)
1       pour i ← 1 à n faire
2               min ← i
3               pour j ← i + 1 à n faire
4                       si A[min] > A[j] alors
5                               min ← j
6                       fin si
7               fin pour
8               si min > i alors
9                       PERMUTER(A, i, min)
10              fin si
11      fin pour
```

3.3 Tri par base

Nom anglais : radix sort - Propriétés : tri interne, stable, ne trie pas sur place

Ce tri n'utilise pas de comparaison entre les clés. Il se base sur la représentation interne de la clé. Ainsi, pour trier des entiers de c chiffres, l'algorithme utilise un tri stable pour trier les clés selon chacun de ces chiffres en commençant par le chiffre des unités. Pour le tri d'entiers, on va choisir le tri par dénombrements comme sous-tri.

Algorithme issu de [1, page 166]

```
TRI-BASE(A, c)
1       pour i ← 1 à c faire
2               utiliser un tri stable pour trier A selon le chiffre i
3       fin pour
```

La complexité de ce tri pour trier n entiers de c chiffres (compris dans l'intervalle 0..k-1) est $\Theta(2c(n+k))$. C'est à dire que la complexité est linéaire.

3.4 Tri par paquets

Nom anglais : bucket sort - Propriétés : tri interne, stable, ne trie pas sur place

Ce tri n'utilise pas de comparaison entre les clés. Comme le tri par dénombrement il se sert de la valeur des clés pour calculer un indice de tableau. L'hypothèse faite sur les données en entrée du tri est qu'elles sont réparties de manière uniforme sur l'intervalle [0, 1[(en pratique, on peut toujours normaliser les données pour les ramener dans cet intervalle. L'hypothèse importante est donc la distribution uniforme). L'idée est de partager cet intervalle [0, 1[en *n* paquets de même taille, puis de placer les données dans ces paquets. Dans un deuxième temps, chacun des paquets est trié à l'aide d'un tri auxiliaire puis les données sont ré-assemblées en parcourant les paquets dans l'ordre. On utilise des listes chaînées pour stocker les clés dans les paquets.

Algorithme issu de [1, page 168]

```
TRI-PAQUETS(A)
1       pour i ← 1 à n faire
2               insérer A[i] dans la liste B[⌊nA[i]⌋]
3       fin pour
4       pour i ← 0 à n - 1 faire
5               trier la liste B[i] via un tri par insertion
6       fin pour
7       concaténer les listes B[0], B[1], ..., B[n - 1] dans l'ordre
```

La complexité de ce tri pour une distribution uniforme est linéaire.

3.5 Tri de Shell

Nom anglais : Shell sort - Propriétés : tri interne, stable, sur place

Le tri de Shell est une variante du tri par insertion. On sait que le tri par insertion fonctionne bien sur des tableaux presque triés. Le tri de Shell utilise un séquence d'entiers, appelée *séquence-h*. Il effectue pour chaque entier de cette séquence un tri par insertion sur les éléments d'indice *i*, *i*+*h*, *i*+2*h*, ... Partant d'une valeur de h très grande pour arriver à 1, l'algorithme organise petit à petit le tableau pour le rendre plus facile à trier lorsque h = 1 (tri par insertion classique).

Algorithme issu de http://en.wikipedia.org/wiki/Shell_sort

```
TRI-SHELL(A, n)
1       h-seq[] = {1391376, 463792, 198768, 86961, 33936, 13776, 4592,1968, 861, 336, 112, 48, 21, 7, 3, 1}
2       pour k ← 0 à 15 faire
3               h ← h-seq[k]
4               pour i ← h à n - 1 faire
5                       v = A[i]
6                       j = i
7                       tant que (j ≥ h) et (A[j - h] > v) faire
8                               A[j] ← A[j - h]
9                               j ← j - h
10                      fin tant que
11                      A[j] ← v
12              fin pour
13      fin pour
```

Tout l'enjeu est de trouver la bonne séquence-h.

	Complexité	Séquence-h
Pratt, 1971	$\Theta\left(n\left(\log n\right)^2\right)$	1 2 3 4 6 9 8 12 18 27 16 24 36 54 81 ...
Papernov-Stasevich, 1965 Pratt, 1971	$\Theta\left(n^{3/2}\right)$	1 3 7 15 31 63 127 255 511 ...
Sedgewick, 1982	$O\left(n^{4/3}\right)$	1 8 23 77 281 1073 4193 16577 ...

4. Quelques exemples d'implémentation et d'utilisation de tris

4.1 Librairie standard Java

Il existe deux principaux algorithmes de tri dans la librairie Java standard (JDK1.4). Ils se trouvent dans la classe utilitaire *java.util.Arrays*.

```
public static void sort(Object[] a)
```

Ce premier algorithme est utilisé pour trier les tableaux d'instances quelconques. Il suffit que les éléments implémentent la méthode *compareTo* de l'interface *Comparable* et qu'ils soient comparables entre eux. Dans ce cas, l'algorithme est un tri fusion amélioré. La documentation explique que ce tri est stable et que les performances en *n*lg*n* sont garanties. En regardant le code source on s'aperçoit qu'un tri par insertion est utilisé pour les tableaux de taille inférieure à 7. De plus si le tableau est déjà trié, l'algorithme ne le re-trie pas.

```
public static void sort(long[] a)
```

Ce deuxième algorithme est utilisé pour trier les tableaux de types primitifs. On regardera la version permettant de trier les entiers de type *long*. Les implémentations pour les autres types primitifs sont très similaires. Il s'agit d'une version améliorée du tri rapide, adaptée d'un algorithme de J. Bentley et D. McIlroy de 1993. Si la taille du tableau est inférieure à 7, un tri par insertion est utilisé. Entre 8 et 40 un tri rapide médiane-de-trois est utilisé, et au delà il s'agit d'un tri rapide pseudo-médiane-de-neuf. Il s'agit de la version vue dans [10] en remplaçant le tri rapide standard par le tri par insertion pour les plus petits ensembles.

4.2 Microsoft Word et Excel

Howard Kaikow propose sur son site web[6] un utilitaire permettant de comparer différentes implémentations d'algorithmes de tri sur des jeux de données aléatoires. Il intègre notamment des appels aux tris de Microsoft Word et Excel. Les données sont placées dans un document Word ou bien dans une feuille de calcul Excel puis sont triées par les différents algorithmes. On obtient le temps en millisecondes nécessaire aux tris. Les résultats se passent de commentaires.

Données			Tris		
Contenant	Type	Nombre	Quick Sort	Excel Sort	Word Sort
Worksheet	Entiers	1000	380	1188	
Worksheet	Entiers	10000	504	1539	
Worksheet	Entiers	25000	530	1634	
Worksheet	Entiers	50000	566	1802	
Worksheet	Chaînes	1000	391	1198	
Worksheet	Chaînes	10000	620	1643	
Worksheet	Chaînes	25000	794	1945	
Worksheet	Chaînes	50000	975	2279	
Document	Entiers	1000	22		83
Document	Entiers	10000	89		966
Document	Entiers	15000	139		1617
Document	Chaînes	1000	16		84
Document	Chaînes	10000	1225		4657
Document	Chaînes	15000	3204		11672

[6] http://www.mv.com/ipusers/standards/

4.3 Oracle

On a vu la méthode utilisée par les bases de données pour effectuer un tri sur un grand ensemble de données (voir chapitre 2.6.2). On va mettre en évidence ce comportement sur une instance Oracle 8.1.7. La taille des données est exprimée en nombre de blocs. Sur cette instance les blocs font 4096 octets. Deux paramètres sont importants pour ce test :

db_block_buffer : espace mémoire cache pour la lecture des blocs de tables
sort_area_size : espace mémoire pour les tris

On va utiliser une table *role* contenant 300 000 enregistrements, et plus particulièrement la colonne *idacteur* qui contient 20 000 clés différentes réparties aléatoirement. La table complète occupe 2205 blocs sur le disque.
On fixe *db_block_buffer* à 65535 blocs. Ainsi la première fois que la table est parcourue elle est entièrement stockée en mémoire cache.
Nous allons faire varier *sort_area_size* et voir comment évolue le temps et le nombre de lectures de blocs nécessaire pour trier l'ensemble des clés *idacteur* de la table *role*. La requête utilisée est :

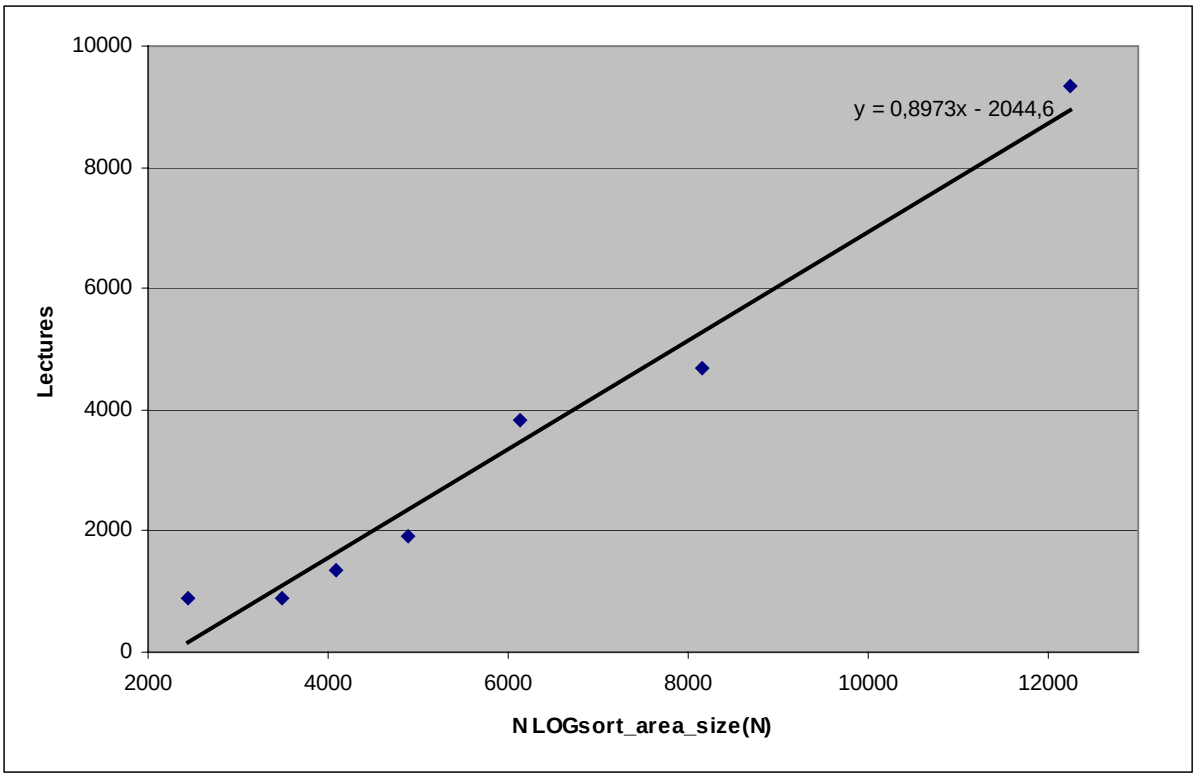
```
select /*+ full(role) */ idacteur from role order by idacteur asc;
```

Cette requête est systématiquement exécutée deux fois pour avoir les résultats avec et sans le rapatriement des blocs de la table. On pourra ainsi isoler les lectures liées au tri. On obtient les résultats suivant (le temps est en ms):

	Exécution 1		Exécution 2	
sort_area_size	lectures	temps	lectures	temps
2	11529	29	9322	25
4	11537	30	9327	26
8	6904	15	4684	12
16	6039	11	3821	8
32	4121	6	1904	4
64	3582	4	1364	2
128	3117	2	900	0,9
1024	3109	2	890	1
2048	2215	1	0	0,9

Le nombre de lectures de blocs de la seconde exécution correspond bien uniquement aux opérations de tri (l'écart avec le nombre de lectures de la première exécution concorde avec la taille de la table). Ce nombre de lectures est en théorie en $O\left(n \log_{sort\_area\_size} n\right)$.

Ceci est vérifié sur le graphe suivant (*sort_area_size* compris entre 4 et 1024), pour lequel on a également fait apparaître la droite issue de la régression linéaire sur ce jeu de test :


10000
8000
6000
4000
2000
0
Lectures
y = 0,8973x - 2044,6
2000
4000
6000
8000
10000
12000
N LOGsort_area_size(N)

5. Simulations sur ordinateur

Le but de ces simulations est de vérifier sur des jeux de test le comportement des algorithmes étudiés. Ces algorithmes ont été implémentés en Java et ont été exécutés sur des tableaux de données générées aléatoirement. Tous les tests ont été fait sur un ordinateur portable équipé d'un processeur Pentium 3GHz, de 512 Mo de mémoire avec un JDK1.4.1 sous Windows XP.

5.1 Détails sur l'implémentation

La mesure du temps en Java n'est pas très précise (~ 10 ms). Les mesures ont donc systématiquement été faites sur des jeux de données suffisamment grands ou bien répétés suffisamment de fois pour palier à ce problème.
Afin de pouvoir facilement changer le type de données à trier, nous avons utilisé une interface *MonitoredData* héritant de l'interface Java classique *Comparable*. En Java la comparaison de deux instances d'une même classe implémentant l'interface *Comparable* se fait grâce à la méthode *compareTo* qui retourne un entier (négatif si la première instance est strictement inférieure à la seconde, positif si elle est strictement supérieure et nul si les deux instances sont égales).
A titre d'exemple voici l'implémentation du tri par insertion :

```
public MonitoredData[] insertionSort(MonitoredData[] tab, int l, int r) {
        MonitoredData key = null;
        int i = 0;
        for (int j = l+1; j <= r; j++) {
                key = tab[j];
                i = j - 1;
                while ((i >= l) && (tab[i].compareTo(key) > 0)) {
                        tab[i + 1] = tab[i];
                        i = i - 1;
                }
                tab[i + 1] = key;
        }
        return tab;
}
```

Il ne reste plus qu'à définir une classe héritant de *MonitoredData* représentant les données que l'on souhaite trier. On remarquera que tous les déplacements de données sont en fait des déplacements de références (pointeurs) et ne sont donc pas coûteux en temps. La plupart des tests qui suivent ont été déroulés sur des entiers à l'aide de la classe *SimpleInteger*.
On a mesuré le temps nécessaire pour effectuer un échange et une comparaison dans un tableau :

temps pour 100 000 000 échanges de *long*	: 547 ms
temps pour 100 000 000 comparaisons de *long*	: 453 ms
temps pour 100 000 000 échanges de *SimpleInteger*	: 844 ms
temps pour 100 000 000 comparaisons de *SimpleInteger*	: 1984 ms

Dans le cas des instances de *SimpleInteger*, le temps d'une comparaison est supérieur au temps d'un échange d'environ 235 % alors que dans le cas du type primitif *long* il est inférieur de 20 % Les écarts entre les deux types de données s'expliquent par les couches objets et les appels de méthodes.

5.2 Les types de distributions d'entiers

Afin de modéliser les différentes organisations de données que l'on est susceptible de rencontrer, nous avons défini 9 distributions possibles. Pour chacune d'entre elles on peut donc générer un jeu de test en fournissant 3 paramètres :

- le nombre d'entiers dans le jeu

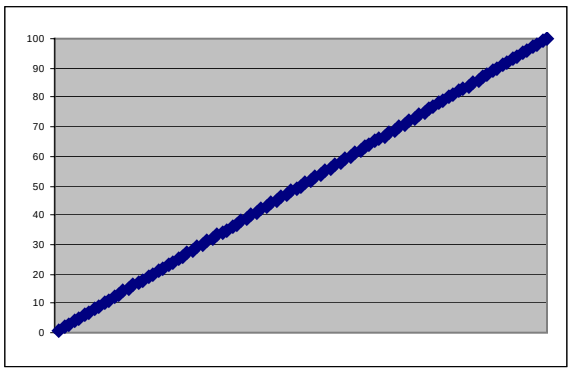
- un entier minimum
- un entier maximum

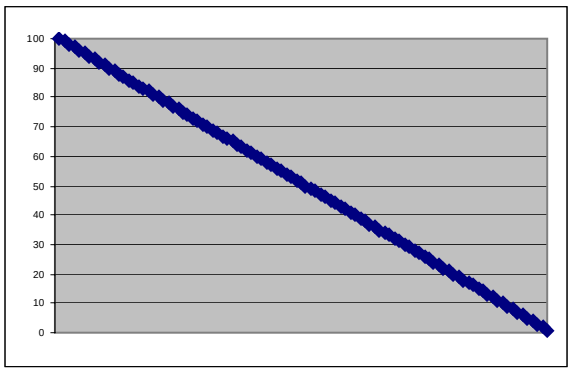
Les jeux de données sont sauvegardés sur fichiers afin d'effectuer les tests sur les différents algorithmes avec les mêmes données.

Les illustrations suivantes ont été générées pour 100 entiers compris entre 1 et 100.

Distribution croissante et décroissante:

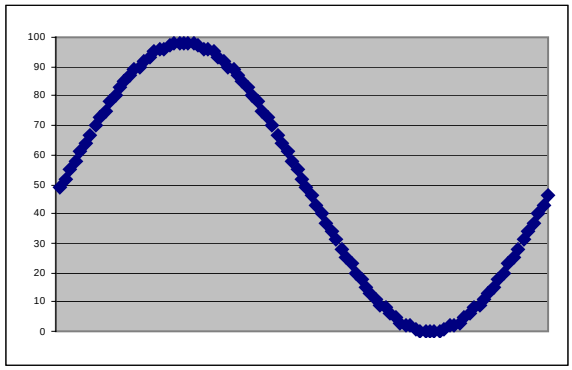
Ce type de distributions fournit des données triées allant du minimum au maximum (ou l'inverse). S'il y a plus d'entiers à générer que ne le permet l'écart entre le minimum et le maximum, des doublons seront ajoutés.





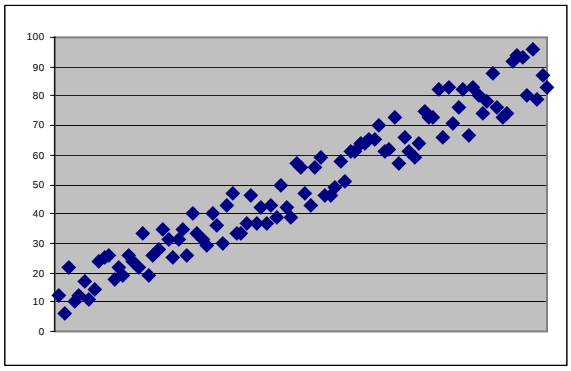
Distribution sinusoïde :

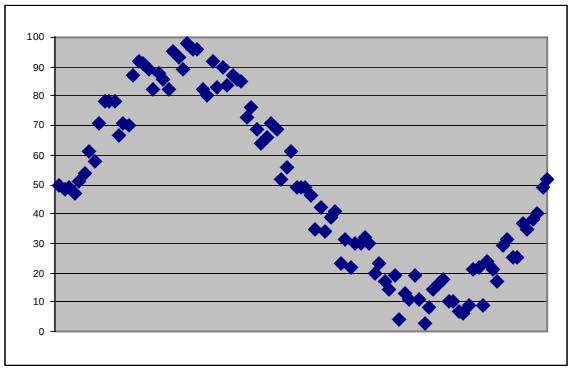
Les données sont générées sous forme de sinusoïde. Il y a donc quelques doublons.



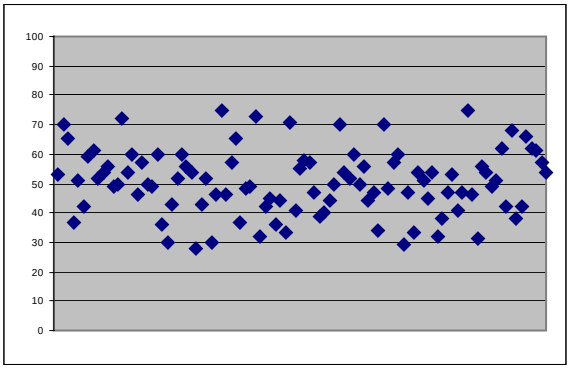
Distributions presque croissante, presque décroissante, presque sinusoïde :

Pour les trois distributions précédentes il y a une variante permettant un écart de x % des valeurs. Il peut donc y avoir des doublons. Les 2 distributions suivantes ont été générées avec un écart de 10 %.

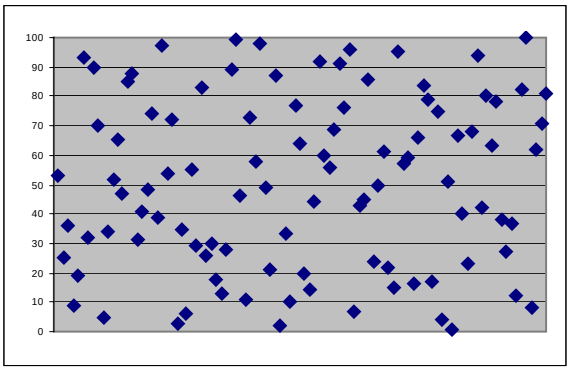




Distribution gaussienne :
Les données sont générées de manière aléatoire, mais selon une distribution gaussienne, c'est à dire qu'elles sont plutôt regroupées autour de la valeur médiane.



Distribution aléatoire et mélangée:
Ces deux distributions sont de type aléatoire. Toutefois, la distribution mélangée assure qu'il n'y a pas de doublon (les entiers entre le minimum et (le minimum + le nombre de données) sont placés dans le tableau puis mélangés de manière aléatoire).



5.3 Résultats généraux

Les 2 tableaux ci-dessous présentent les temps d'exécution et le nombre de comparaisons obtenus en faisant la moyenne sur 100 jeux de données (minimum: 1, maximum: 10 000), chacun étant exécuté 10 fois. Il manque certaines valeurs pour le tri rapide sur les distributions croissantes et décroissantes. Ceci est dû à un débordement de la pile créé par la trop grande profondeur des appels récursifs liée à un partitionnement déséquilibré (voir chapitre 2.4).

Les résultats théoriques sont vérifiés :
- le tri par insertion fonctionne bien sur les tableaux presque triés
- le nombre de comparaisons du tri par insertion correspond aux valeurs théoriques à 99 %
- le nombre de comparaisons du tri rapide correspond aux valeurs théoriques à 99 %
- le tri rapide aléatoire permet de lisser les comportements selon les distributions par rapport à la version standard
- la version améliorée du tri rapide avec coupure pour M=7 permet un gain de temps de 5%
- le tri fusion est insensible à la distribution des données

On remarque que le tri par tas est globalement 50% moins performant que le tri rapide. Le tri de Java est le plus performant.

Tri par insertion																		
distribution	croissante		décroissante		presque croissante		presque décroissante		sinusoide		presque sinusoide		aléatoire		mélangée		gauss	
n	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp
1 000	0	999	23,45	499 500	1,72	39 718	19,83	461 429	17,20	375 240	16,73	374 434	11,57	248 177	11,25	253 004	10,78	255 021
2 500	0	2 499	132,67	3 123 750	15,93	244 833	125,63	2 882 814	112,37	2 344 280	107,82	2 338 778	81,88	1 567 268	76,39	1 558 008	75,30	1 575 645
5 000	0	4 999	519,52	12 497 500	32,96	976 368	399,70	11 520 263	392,36	9 375 682	348,76	9 350 420	215,94	6 258 105	217,35	6 259 885	244,70	6 212 282
10 000	0	9 999	2 233,26	49 995 000	131,87	3 913 141	1 799,52	46 099 470	1 689,86	37 499 531	1 609,55	37 408 919	1 129,54	25 053 037	1 018,74	25 005 426	1 050,95	24 971 210

Tri à bulles																		
distribution	croissante		décroissante		presque croissante		presque décroissante		sinusoide		presque sinusoide		aléatoire		mélangée		gauss	
n	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp
1 000	15,62	499 500	32,82	499 500	17,19	499 500	31,25	499 500	33,58	499 500	28,13	499 500	29,86	499 500	23,77	499 500	29,39	499 500
2 500	92,04	3 123 750	196,86	3 123 750	106,56	3 123 750	184,84	3 123 750	191,26	3 123 750	170,32	3 123 750	161,28	3 123 750	140,16	3 123 750	160,48	3 123 750
5 000	313,26	12 497 500	800,00	12 497 500	397,32	12 497 500	761,24	12 497 500	694,05	12 497 500	694,83	12 497 500	608,30	12 497 500	554,37	12 497 500	661,85	12 497 500
10 000	1 456,88	49 995 000	3 163,12	49 995 000	1 641,25	49 995 000	3 047,50	49 995 000	2 975,18	49 995 000	2 750,32	49 995 000	2 779,70	49 995 000	2 383,92	49 995 000	3 027,49	49 995 000

Tri fusion																		
distribution	croissante		décroissante		presque croissante		presque décroissante		sinusoide		presque sinusoide		aléatoire		mélangée		gauss	
n	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp
1 000	1,55	9 976	1,58	9 976	1,56	9 976	1,56	9 976	1,57	9 976	1,56	9 976	1,57	9 976	1,57	9 976	1,56	9 976
2 500	3,12	28 404	3,13	28 404	3,12	28 404	3,10	28 404	3,13	28 404	3,12	28 404	3,10	28 404	3,12	28 404	3,12	28 404
5 000	6,24	61 808	6,27	61 808	6,25	61 808	6,23	61 808	7,35	61 808	6,23	61 808	6,39	61 808	7,81	61 808	7,67	61 808
10 000	14,06	133 616	12,98	133 616	14,05	133 616	13,26	133 616	12,50	133 616	14,06	133 616	14,04	133 616	14,06	133 616	12,65	133 616

Tri Java (tri fusion modifié)																		
distribution	croissante		décroissante		presque croissante		presque décroissante		sinusoide		presque sinusoide		aléatoire		mélangée		gauss	
n	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp
1 000	0,00	999	0,00	5 763	0,00	8 016	0,00	8 086	0,00	4 378	0,15	8 006	0,00	8 997	0,00	9 010	0,80	9 001
2 500	1,53	2 499	1,71	16 739	1,73	23 437	1,57	23 513	1,55	12 108	1,53	23 400	2,98	25 870	2,34	25 881	3,15	25 899
5 000	1,60	4 999	3,26	35 979	4,68	52 057	4,69	52 030	3,11	25 462	4,67	51 726	4,86	56 786	6,26	56 763	6,25	56 767
10 000	3,10	9 999	9,35	76 959	11,09	114 140	10,93	114 302	5,01	53 390	10,92	113 673	10,49	123 490	10,63	123 514	11,11	123 555

Tri rapide standard																		
distribution	croissante		décroissante		presque croissante		presque décroissante		sinusoide		presque sinusoide		aléatoire		mélangée		gauss	
n	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp
1 000	37,34	499 500	19,18	499 500	1,57	38 351	1,58	18 006	4,69	85 456	1,34	14 291	0,16	10 790	0,00	11 110	0,93	11 188
2 500	204,67	3 123 750	121,73	3 123 750	12,82	142 999	4,46	54 179	35,91	496 572	4,68	45 923	3,13	31 535	3,12	32 181	3,13	31 693
5 000	845,00	12 497 500	450,85	12 497 500	19,14	389 859	9,53	126 502	94,54	1 959 728	10,06	116 188	4,86	69 873	6,27	70 910	6,26	72 158
10 000					52,74	1 084 114	17,90	292 387	368,11	7 784 402	19,31	289 374	11,73	155 587	12,50	152 510	12,34	159 884

Tri rapide aléatoire																		
distribution	croissante		décroissante		presque croissante		presque décroissante		sinusoide		presque sinusoide		aléatoire		mélangée		gauss	
n	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp
1 000	0,00	10 986	0,16	10 980	1,55	10 996	1,58	10 967	0,94	10 915	1,52	11 073	1,55	10 948	0,63	11 070	1,50	10 934
2 500	3,12	31 985	3,28	31 897	3,14	32 412	3,11	32 349	3,12	32 068	3,13	31 760	3,13	32 021	3,11	31 756	3,12	31 971
5 000	6,25	71 177	6,25	70 915	6,26	70 803	6,24	70 932	6,24	70 758	6,23	70 695	6,40	71 181	6,90	70 690	6,56	71 038
10 000					12,81	156 068	12,95	156 001	12,66	158 502	13,12	156 441	13,27	156 106	12,96	157 629	12,35	157 261

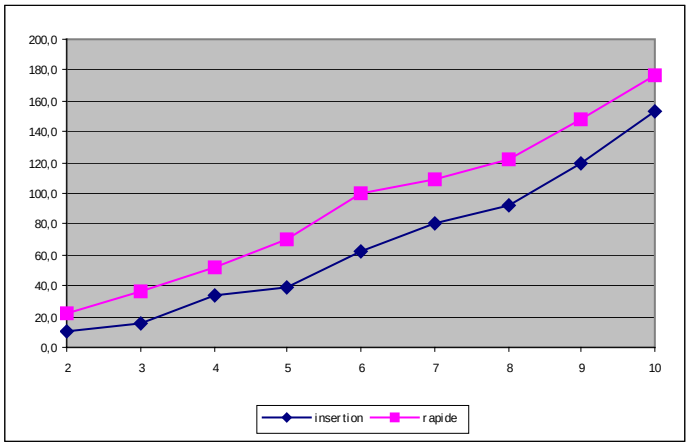
Tri rapide standard + tri insertion pour M=7																		
distribution	croissante		décroissante		presque croissante		presque décroissante		sinusoide		presque sinusoide		aléatoire		mélangée		gauss	
n	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp
1 000	37,51	499 485	21,87	499 495	1,57	38 380	1,56	18 030	4,70	85 458	1,41	14 374	0,15	10 823	0,00	11 179	0,78	11 264
2 500	212,81	3 123 735	143,91	3 123 745	13,75	143 116	4,29	54 285	38,30	496 558	4,70	46 044	3,14	31 656	3,13	32 319	2,97	31 757
5 000	870,91	12 497 485	538,91	12 497 495	19,92	389 927	9,22	126 567	96,87	1 959 634	10,16	116 278	5,00	69 957	6,25	71 259	6,10	71 879
10 000					54,22	1 083 571	17,74	291 790	386,41	7 784 039	19,46	288 933	11,88	155 206	11,89	153 215	11,72	157 429

Tri par tas																		
distribution	croissante		décroissante		presque croissante		presque décroissante		sinusoide		presque sinusoide		aléatoire		mélangée		gauss	
n	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp	ms	comp
1 000	1,58	17 583	1,53	15 965	1,25	16 926	1,1	16 234	1,25	16 616	1,57	16 737	1,85	16 879	1,57	16 832	1,56	16 862
2 500	4,84	51 032	3,59	46 694	4,86	49 228	4,54	47 159	4,69	48 190	4,54	48 627	5	48 809	4,69	48 846	4,68	48 842
5 000	10,93	112 126	10,14	103 227	11,08	108 505	9,99	104 338	9,69	106 312	10,66	107 249	10,93	107 708	10,63	107 684	10,79	107 733
10 000	17,97	244 460	17,36	226 682	18,12	236 974	17,02	228 643	17,52	232 324	17,8	234 532	17,34	235 374	17,97	235 397	17,97	235 326

5.4 Résultats sur de petits tableaux

Les temps ci-dessous correspondent à 100 000 exécutions (moyenne sur 10 jeux différents de distribution mélangée) :

n	insertion	rapide
2	11,0	21,9
3	15,7	36,0
4	34,4	51,6
5	39,1	70,4
6	62,3	99,8
7	79,9	109,5
8	92,1	121,9
9	118,9	148,3
10	153,0	176,7
20	481,3	482,8
50	2 643,6	1 603,1



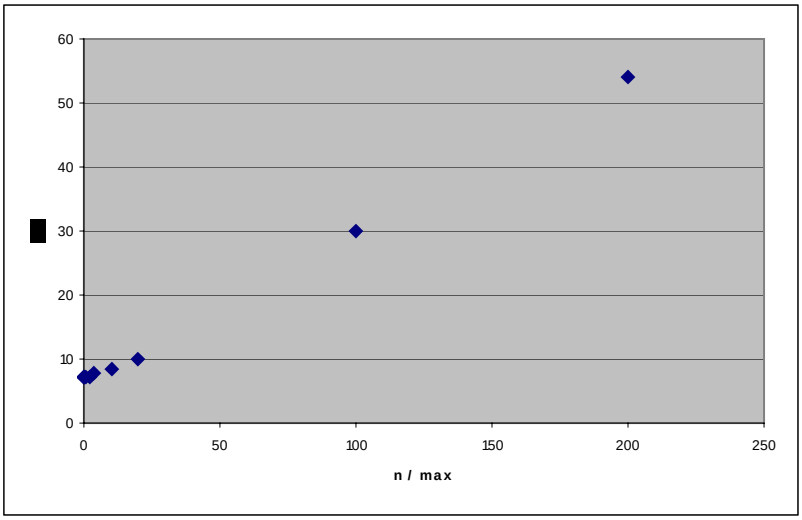
On voit que pour de petits tableaux le tri par insertion est plus rapide que le tri rapide standard. Ceci est vrai jusqu'à $n = 20$ environ.

5.5 Sensibilité aux doublons du tri rapide

Puisque toutes les études sur le tri rapide ont été faites en prenant comme hypothèse que les clés étaient uniques dans la distribution, nous avons fait tourner quelques jeux sur lesquels nous faisant varier le nombre de fois qu'une clés est présente. On a pris des distribution aléatoire de 10 000 clés pour lesquelles on a fait varier le maximum. Le tri rapide standard a été utilisé. Les performances se dégradent dès que des doublons apparaissent.

max	n / max	ms	comp
50	200	54,186	1070127,8
100	100	29,889	587440,1
500	20	10,079	216227,2
1000	10	8,498	174022,2
2500	4	7,842	162161,5
5000	2	7,237	160409,7

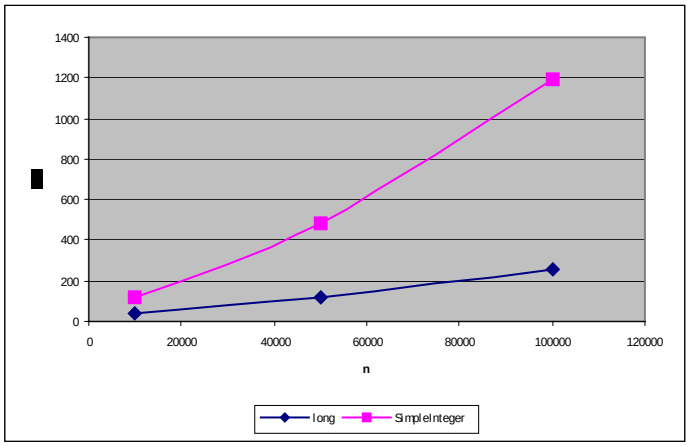
max	n / max	ms	comp
10000	1	7,173	156970,5
50000	0,2	7,126	156636,4
100000	0,1	7,313	158061,7
500000	0,02	7,033	155158,9
1000000	0,01	7,219	155492,3



5.6 Comparaison des deux tris de Java

Les temps ci-dessous correspondent à 10 exécutions (moyenne sur 10 jeux différents de distribution mélangée) :

n	long	SimpleInteger
10 000	39	115,4
50 000	121,9	487,6
100 000	256,4	1 196,9



Les différences s'expliquent par l'écart entre le temps de comparaison de deux *SimpleInteger* et le temps de comparaison de deux *long* (voir 5.1).

Conclusion

A travers les différentes analyses théoriques et les tests déroulés sur ordinateur, on peut se rendre compte de la diversité des algorithmes de tri. Cette étude a permis de mettre en avant un certain nombre de paramètres qu'il faut bien prendre en compte au moment de choisir un algorithme de tri. Un point important est de bien définir le modèle des données à trier (type, distribution, bornes), ainsi que l'utilisation exacte qui sera faite de l'algorithme (librairie générique, utilisation pour un problème très spécifique).

Les deux principaux tris par comparaisons sont le tri fusion et le tri rapide. Le premier est sûr de partitionner de manière équilibré à chaque étape, mais il déplace les données moins rapidement vers leur position finale, à l'inverse du second. Les optimisations sur ces deux tris ont pour but de les rendre optimaux au regard de la limite théorique sur ce type de tri.

Globalement les deux algorithmes de tri implémentés dans la librairie Java sont performants étant donné qu'ils sont génériques et n'ont pas d'hypothèses particulières sur les données. Pour une utilisation plus poussée, il faut penser à regarder les versions optimisées dans les rapports de recherche. Il ne faut pas hésiter non plus à passer de la version récursive à la version itérative des algorithmes "diviser pour régner".

Lorsque c'est possible, il ne faut pas oublier les tris linéaires.

Un certain nombre de pistes n'ont pas été abordées mais pourraient faire l'objet d'une étude plus poussée :

- algorithmes de tri sur des listes chaînées plutôt que sur des tableaux
- les algorithmes spécifiques de tri de chaînes de caractères
- les tris externes sur bandes magnétiques
- lien entre le tri et la recherche de clés
- les algorithmes de tri sur machines multi-processeurs

Si on en juge par la fréquence des parutions des laboratoires informatiques sur la thématique du tri, il semble que, malgré son ancienneté, ce sujet soit encore amené à voir des optimisations, et peut être de nouveaux algorithmes.

Bibliographie

[1] T. CORMEN, C. LEISERSON, R. RIVEST et C. STEIN, *Introduction à l'algorithmique 2ème édition*, Paris, éditions Dunod, 2002, chapitres 1 à 8

[2] D. BEAUQUIER, J. BERSTEL et P. CHRETIENNE, *Eléments d'algorithmique*, Paris, éditions Masson, 1992, chapitres 1 et 5

[3] R. FAURE, B. LEMAIRE, C. PICOULEAU, *Précis de recherche opérationnelle 5ème édition*, Paris, éditions Dunod, 2000, chapitre 2

[4] P. RIGAUX et M. SCHOLL, *Cours de Bases de Données Aspects Systèmes(draft)*, septembre 2001

[5] S. SEIDEN, *Theoretical Computer Science Cheat Sheet*, 1994, http://www.tug.org/texshowcase/cheat.pdf

[6] A. AGGARWAL et J. S. VITTER, *The input/output of sorting and related problems*, INRIA RR-0725, septembre 1987

[7] R. SEDGEWICK et P. FLAJOLET, *Introduction à l'analyse des algorithmes*, Paris, International Thomson Publishing France, 1996, chapitre 1

[8] R. SEDGEWICK, *Algorithmes en langage C*, Paris, éditions Dunod, 2001, chapitres 6, 9, 12 et 13

[9] R. SEDGEWICK et J. BENTLEY, *Quicksort is optimal*, Stanford University, 2002, http://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf

[10] M. DURAND, *Asymptotic analysis of an optimized quicksort algorithm*, INRIA Rocuencourt, 2003, http://algo.inria.fr/durand/

[11] S. BHUTORIA et G. KONJEVOD, *Quickening quick-sort*, 2003, http://www.public.asu.edu/~sbhutori/docs/reas.pdf