G. Buss/PIX, S.A.

L'art du tri

JEAN-PAUL DELAHAYE

La loi du moindre effort dans les méthodes de tri de données ou de cartes. Gagnez un temps précieux !

Pour vérifier qu'un jeu est complet ou identifier les cartes absentes, pour préparer certains tours de prestidigitation ou un tournoi de bridge, il faut trier les cartes. Les problèmes posés par ces tris ressemblent à ceux que doivent traiter les informaticiens pour classer des données. Pour les résoudre, une science inattendue de l'économie des gestes et du travail a été mise en œuvre.

Les mathématiciens ont compris il y a 40 ans que trier n'était pas si simple et, depuis, ils ont étudié comment préserver le précieux temps de calcul des ordinateurs. Parmi des dizaines de méthodes proposées, certains algorithmes remarquables de tri informatique s'adaptent au classement des cartes et illustrent que le plus rapide n'est pas le plus simple. Mais, comme on le verra aussi, le classement des cartes a ses propres subtilités.

LE TRI PAR INSERTION

Le tri *par insertion* est sans doute le plus naturel de tous les tris à la main, et c'est d'ailleurs celui que la plupart des gens utilisent pour classer leurs cartes quand ils ramassent leur main au bridge ou quand ils doivent trier alphabétiquement des noms, des fiches, ou des livres.

L'algorithme de tri par insertion est le suivant : on prend les cartes une à une, et l'on compose un nouveau paquet en plaçant chaque carte qu'on introduit à la place qu'elle doit avoir parmi celles qui sont déjà placées. À chaque instant du tri par insertion, une partie du paquet est parfaitement classée et l'autre encore désordonnée.

On mesure la complexité d'un tri en comptant le nombre de comparaisons entre deux cartes (ou deux nombres s'il s'agit de nombres) qu'on doit faire pour classer l'ensemble. Plus ce nombre est grand, moins bon est le tri ; plus il est petit, meilleur il est.

Étudions ce qui se passe pour le tri par insertion. Quand on place la première carte, on n'effectue aucune comparaison. Quand on place la seconde carte, on effectue une comparaison : il faut savoir si la carte qu'on introduit est plus faible ou plus forte que celle qui est déjà placée. Quand on place la troisième carte, on effectue deux comparaisons : la carte nouvelle est comparée avec chacune des deux cartes déjà en place pour savoir si elle doit se placer devant les deux, au milieu ou derrière. Et ainsi de suite : le placement de la carte numéro p peut nécessiter $p - 1$ comparaisons. Au total donc, on aura effectué au plus $1 + 2 + \ldots + (n - 1) = n\,(n - 1)/2$ comparaisons quand la n -ième carte aura été placée. J'ai bien précisé «au plus», car, quand vous placez la carte numéro p, vous n'avez vraiment besoin d'effectuer $p - 1$ comparaisons que si sa place est en dernier (vous pouvez placer la carte p dès que vous rencontrez une carte plus forte).

Pour 10 données, cela fait au plus 45 comparaisons ; pour 100 données, au plus 4 950 ; pour 1 000 données, au plus 499 500. Pour obtenir le nombre moyen de comparaisons, il faut diviser ces nombres par 2.

L'augmentation du travail à faire (dans le pire cas, ou en moyenne) est une fonction polynôme de degré deux du nombre de cartes. On dit que l'algorithme est quadratique et en pratique cela signifie qu'il devient rapidement inutilisable.

LE TRI-RAPIDE

Le *tri-rapide* est plus difficile à effectuer, mais le jeu en vaut la chandelle, car il nécessite, au total, bien moins de comparaisons. Son avantage par rapport au tri par insertion est très net quand le nombre de cartes à trier augmente. Nous décrivons cet algorithme par étapes.

Étape 1. On prend la première carte (qu'on appellera carte pivot), on la place sur la table au milieu et en haut. La carte pivot constitue la ligne 1. On forme alors deux paquets en comparant toutes les cartes restantes avec la carte pivot : les cartes plus faibles sont mises en paquet à gauche sur la ligne 2, les plus fortes sont mises à droite sur la ligne 2 *(voir la figure 2).*

Étape 2. On prend toutes les cartes du paquet de gauche de la ligne 2, sauf la première qui sert comme précédemment de carte pivot et qu'on laisse sur la ligne 2. On sépare comme précédemment ce paquet en deux nouveaux paquets plus petits qu'on place sur la troisième ligne. Le paquet de droite de la ligne 2 est traité de la même manière. Il y a donc maintenant une carte sur la ligne 1, deux sur la ligne 2, et quatre petits paquets sur la ligne 3.

Étape 3. On retire une carte de chaque paquet de la ligne 3 qui y reste et l'on sépare le reste de chaque paquet comme précédemment en deux, ce qui donne en tout 8 paquets qu'on place sur la ligne 4.

On continue de même jusqu'à n'avoir plus que des cartes seules, chaque paquet ayant été totalement fractionné. Les cartes sont alors disposées en une structure ressemblant à un arbre à l'envers. À chaque carte de l'arbre sur la ligne n sont associées deux cartes au plus de la ligne $n + 1$, qu'on appellera ses fils gauche et droit ; la carte sera bien sûr appelée la carte père de ses deux fils (le mouvement *Politically Correct,* aux États-Unis, oblige les informaticiens à parler de cartes filles et de cartes mères, mais en France nous pouvons garder une certaine liberté de langage).

Quand toutes les cartes sont ainsi disposées en un arbre à l'envers, on les ramasse en «repliant l'arbre» jusqu'à n'avoir plus qu'un seul paquet (qui est alors classé) : on repère un paquet sur une ligne n n'ayant aucun paquet asso-

cié sur la ligne $n + 1$ (le paquet n'a pas de fils), on le place sur le paquet correspondant (son père) de la ligne $n - 1$ s'il s'agit d'un fils gauche, et en dessous s'il s'agit d'un fils droit.

La complication de ce tri inventé en 1962 par C. Hoare en vaut vraiment la peine, car le nombre de comparaisons qu'on doit réaliser pour classer n cartes (ou données) est cette fois en moyenne à peu près $n.\log_2(n)$. Dans cette formule, il faut arrondir le logarithme en base 2 de n à l'entier supérieur (rappelons que $\log_2(2) = 1$, $\log_2(4) = 2$, $\log_2(8)=3$, etc.) En définitive, pour 10 données, il faut environ 40 comparaisons, pour 100 données, environ 700, et pour 1 000, environ 10 000, ce qui est sensiblement moins qu'avec le tri par insertion.

La justification de ce $n.\log_2(n)$ est la suivante. D'une part, pour construire chaque ligne, il faut comparer chaque carte restante à une carte de la ligne précédente. Il faut donc faire environ n comparaisons à chaque construction de ligne. D'autre part, le nombre de lignes est environ $\log_2(n)$, car il y a 1 carte au niveau 1, 2 cartes au niveau 2, 4 au niveau 3, 2^{n-1} au niveau n. Ce calcul n'est qu'une évaluation, car certains paquets s'épuisent plus vite que d'autres, mais l'idée expliquée peut être rendue parfaitement rigoureuse.

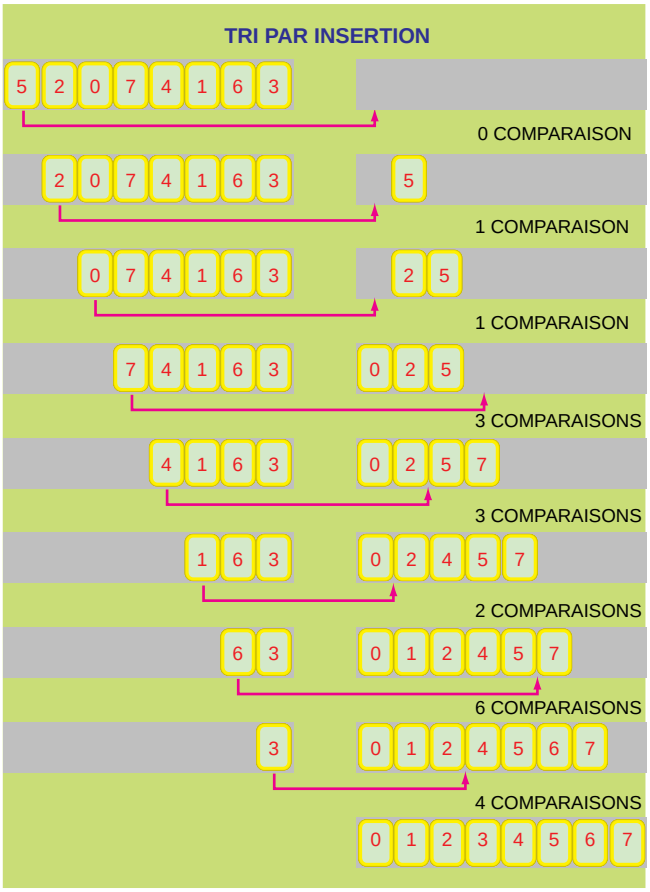
Dans les mauvais cas (quand la séparation par le pivot est systématiquement déséquilibrée), le tri-rapide a exactement une complexité de $n\,(n-1)/2$ comparaisons, c'est-à-dire la complexité du tri par insertion : ce qui distingue les deux tris, c'est le comportement en moyenne, qui est toujours quadratique pour le tri par insertion, alors qu'il est en $n.\log_2(n)$ pour le tri-rapide.

Assez paradoxalement, les cas où le tri-rapide est le plus lent se présentent quand le paquet initial est déjà totalement ou partiellement trié. Statistiquement, ces mauvais cas se présentent très rarement (d'où la moyenne en $n.\log_2(n)$). Cependant il se trouve que les données qu'on doit trier ne sont bien souvent pas statistiquement quelconques et qu'en pratique elles sont au contraire souvent partiellement ou totalement triées : se reposer sur la rareté statistique (très différente de la rareté pratique) des mauvais cas est donc un mauvais calcul (que certains programmeurs mal informés font pourtant).

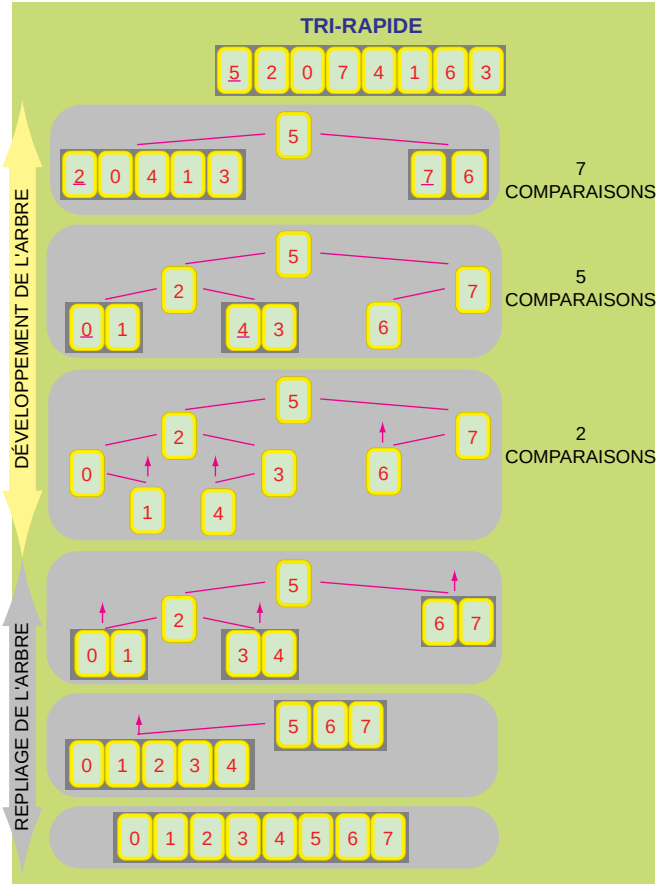
Il existe heureusement un moyen simple d'arranger les choses et qui permet de considérer en définitive que le tri-rapide est un bon algorithme : il faut mélanger les données avant de les trier ! C'est aujourd'hui ce qu'on recommande à ceux qui, en informatique, veulent utiliser le tri-rapide. Avec cette précaution, le tri-rapide est alors très efficace, et bien meilleur que le tri par insertion.

LE MIEUX QUE L'ON PUISSE FAIRE

C'est même un algorithme qu'on ne peut pas espérer améliorer, car $n.\log_2(n)$ est la borne la meilleure qu'on puisse atteindre : aucun algorithme de tri de données n'utilisant que des comparaisons ne peut en faire moins de $n.\log_2(n)$ en moyenne. La justification de ce résultat est intéressante. Notons qu'il y a $n!$ classements différents des cartes et que pour les trier il faut nécessairement les discriminer les uns des autres. Avec une comparaison, on ne peut discriminer que 2 classements ; avec 2 comparaisons, on ne peut discriminer que $4 = 2^2$ classements, etc. Donc, pour discriminer les $n!$ classements possibles



1. Le tri par insertion est naturel et simple à programmer. Malheureusement, ce n'est pas un tri efficace, car, pour trier n données, il lui faut en moyenne effectuer $n^2/4$ comparaisons.



2. Le tri-rapide est beaucoup plus efficace, même s'il est plus complexe à programmer : pour trier n données, il lui faut, en moyenne, effectuer $n.\log_2 n$ comparaisons.