Leçon 903 : Exemples d'algorithmes de tri. Correction et complexité

Julie Parreaux

2018 - 2019

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[2] Cormen, *Algorithmique.*
[3] Froidevaux, Gaudel et Soria, *Types de données et algorithmes.*

Développements de la leçon

Le tri par tas — Le tri topologique

Plan de la leçon

Motivation

Défense

Le problème de tri est considéré par beaucoup comme un problème fondamental en informatique. Mais pourquoi trier ?

— Le problème de tri peut être inhérent à l'application : on cherche très souvent à classer des objets suivant leurs clés, comme lors de l'établissement des relevés bancaires.

— Le tri est donc souvent utilisé en pré-traitement dans de nombreux domaines de l'algorithmique : la marche de Jarvis (enveloppe convexe), l'algorithme du peintre (rendu graphique : quel objet je dois afficher en dernier pour ne pas l'effacer par d'autres ?), la recherche dans un tableau (dichotomie), l'algorithme de Kruskal (arbre couvrant minimal) ou encore l'implémentation de file de priorité.
— Le tri a un intérêt historique : de nombreuses techniques ont été élaborées pour optimiser le tri.
— Le problème de tri est un problème pour lequel on est capable de trouver un minorant (en terme de complexité).
— L'implémentation d'algorithme de tri fait apparaître de nombreux problèmes techniques que l'on cherche à résoudre via l'algorithmique.

Dans cette leçon, nous effectuons deux hypothèses importantes : les éléments à trier tiennent uniquement en mémoire vive et l'ordre sur ces éléments (sous lequel on tri) est total.

Ce qu'en dit le jury

Sur un thème aussi classique, le jury attend des candidats la plus grande précision et la plus grande rigueur.

Ainsi, sur l'exemple du tri rapide, il est attendu du candidat qu'il sache décrire avec soin l'algorithme de partition et en prouver la correction en exhibant un invariant adapté. L'évaluation des complexités dans le cas le pire et en moyenne devra être menée avec rigueur : si on utilise le langage des probabilités, il importe que le candidat sache sur quel espace probabilisé il travaille.

On attend également du candidat qu'il évoque la question du tri en place, des tris stables, des tris externes ainsi que la représentation en machine des collections triées.

1 Le problème de tri

— Problème du tri [1, p.122] les objets sont triés selon une clé (données satellites ou non)
  entrée n éléments $a_1, \dots, a_n$ d'un ensemble E totalement ordonné
  sortie une permutation des éléments $\sigma \in \mathfrak{S}_n$ telle que $s_{\sigma(1)} \leq \cdots \leq a_{\sigma(n)}$
— *Hypothèses* : tri interne (tout est en mémoire vive) et ordre total
— *Définition* : tri stable [3, p.307]
— *Exemples* de tri dont un stable et l'autre non
— *Définition* : tri en place [2, p.136]
— Critère de comparaison des algorithmes de tri : complexité temporelle (pire cas / en moyenne) ; complexité spatiale ; stabilité ; caractère en place.

Tri	Pire cas	En moyenne	Spatiale	Stable	En place
Sélection	$O(n^2)$	$O(n^2)$	$O(1)$	☑	☑
Insertion	$O(n^2)$	$O(n^2)$	$O(1)$	☑	☑
Fusion	$O(n \log n)$	$O(n \log n)$	$O(n)$	☒	☒
Tas	$O(n \log n)$	–	$O(1)$	☒	☑
Rapide	$O(n^2)$	$O(n \log n)$	$O(1)$	☒	☑
Dénombrement	$O(k+n)$	$O(k+n)$	$O(k)$	☑	☒
Base	$O(d(k+n))$	$O(d(k+n))$	$O(dk)$	☑	☒
Paquets	$O(n^2)$	$O(n)$	$O(n)$	☒	☒

2 Les tris par comparaison

Ici la complexité de nos algorithmes peuvent être calculer en nombre de comparaisons effectuées.

— *Définition* : Tri par comparaison [2, p.178] On s'autorise alors uniquement cinq tests sur les données à l'entrée : $=, <, >, \leq, \geq$.

— *Théorème* : borne optimale des tris par comparaisons $\Omega(n \log n)$ [2, p.179] On peut faire un premier classement des tris : ceux qui sont optimaux en moyenne, au pire cas ou pas du tout.

2.1 Les tris naïfs [3, p.310]

On commence par étudier les tris naïfs : ceux qui ne mettent en place aucun paradigme de programmation ni structures de données élaborées.

Tri par sélection

— *Principe* : on cherche le minimum des éléments non triés et on le place à la suite des éléments triés

— Algorithmes classique (algorithme 1) (se fait de manière récursive en cherchant le minimum de manière itérative à chaque fois) et tri à bulle (algorithme 3) le tri à bulle est un des tri par sélection le plus simple à programmer : il se base sur l'idée que l'on part de la fin de la liste et qu'on fait remonter chacun des éléments tant qu'il est plus petit que celui devant lui. On peut également parler du tri boustrophédon qui est un tri à bulle dans les deux sens (on fait descendre les éléments les plus lourds et remonter les plus légers).

— *Complexité* : $O(n^2)$

— *Propriétés* : stable et en place

Tri par insertion (le tri par insertion est aussi appeler la méthode du joueur de carte)

— *Principe* : On insère un à un les éléments parmi ceux déjà trié.

— Algorithme 4

— *Complexité* : $O(n^2)$

— *Propriétés* : stable et en place

— *Remarques* : très efficace sur des petits tableau ou sur des tableau presque trié. Java implémente ce tri pour des tableau de taille inférieure ou égale à 7.

2.2 Diviser pour régner

On présente maintenant des algorithmes de tri basé sur le paradigme diviser pour régner : le premier découpe simplement le tableau de départ (tri fusion) tandis que le second combine facilement les deux sous tableau triés (tri rapide).

Tri fusion [2, p.30]

— *Principe* : On découpe l'ensemble des données en deux sous-ensembles de même taille que l'on tri séparément. Ensuite, nous combinons ces deux ensembles en les entrelaçant.

— Algorithme 9

— *Complexité* (pire cas et moyenne) : $O(n \log n)$

— *Application* : Calcul de jointure dans le cadre des bases de données

Tri rapide [2, p.157] Cet exemple est intéressant d'un point de vu du tri puisqu'il possède de bonne performance. De plus, c'est un algorithme soumis au principe de diviser pour régner dont on ne connaît pas la taille des sous-problème à priori : c'est un bon contre-exemple au Master theorem.

— *Principe* : On sépare l'ensemble des données en deux sous-ensembles tels que le premier sous-ensemble ne contient que des valeurs inférieures à un pivot et que le second que les valeurs supérieures à ce pivot. On tri ensuite chacun de ces deux sous-ensembles et on les combines en les concaténant.
— Algorithme 6
— *Complexité* dans le pire cas : $O(n^2)$; en moyenne $O(n \log n)$
— *Propriété* : en place ; le plus rapide en pratique

Il existe des tris mixtes qui en fonctions de l'ensemble des données (taille, caractère trié) que l'on présente choisi l'un ou l'autre des algorithmes de tri.

2.3 Structure de données [2, p.140]

C'est un algorithme de tri qui se base sur les propriétés d'une structure de données bien précise : le tas. Elle permet de gérer les données. Le terme tas qui fut d'abord inventé dans ce contexte est aujourd'hui également utilisé dans un contexte d'exécution d'un programme : c'est une partie de la mémoire qu'utilise un programme. Un programme lors qu'il s'exécute utilise une mémoire de pile (qui donne les instructions suivantes à faire) et un tas (qui contient les valeurs des variables, de la mémoire auxiliaire pour faire tourner le programme). Ces deux mémoires grandissent l'une vers l'autre (l'erreur de *stack overflow* arrive quand la pile rencontre le tas).

— *Définition* : un tas
— *Principe* : On construit un tas avec les éléments de l'entrée et on extrait le maximum de ce tas itérativement.
— Algorithme, correction et complexité DEV
— *Propriétés* : en place
— *Application* : file de priorité

Le tri par tas est un tri en place et de complexité $O(n \log n)$ en moyenne. C'est donc un des meilleurs tri par comparaison que l'on possède.

3 Les tris linéaires

On fait des hypothèses sur l'entrée de nos algorithmes afin d'obtenir un tri linéaire. De plus, ces algorithmes fond appel à d'autres opérations que les comparaisons : on fait abstraction de la borne de minimalité.

Tri par dénombrement [2, p.180]

— *Hypothèse* : on suppose que les valeurs de l'entrées sont comprises entre 0 et $k \in \mathbb{N}$ fixé.
— *Principe* : déterminer pour chaque élément x combien sont inférieur à lui.
— *Méthode* : dans un tableau C de longueur k : $C[i]$ contient le nombre de clés $\leq i$.
— Algorithme 10 et exemple (Figure 1)
— *Propriétés* : tri stable Faire attention si plusieurs valeurs sont égales.
— *Complexité* : $O(n+k)$ si $k = O(n)$ on retrouve bien le linéaire

Tri par paquets [2, p.185]

— *Hypothèse* : les clés sont uniformément distribuées sur $[0, 1[$.

— *Principe* : on divise $[0, 1[$ en n intervalles de même taille : on distribue les entrées dans ces intervalles et on les tri par insertion dans chaque intervalle avant de les concaténer.

— Algorithme 11

— *Complexité espérée* : $O(n)$.

Tri par base [2, p.182]

— *Hypothèse* : d sous-clés entières bornées muni d'un poids de plus en plus fort

— *Principe* : trier les sous-clé du poids le plus faible à l'aide d'un tri par dénombrement sa stabilité est essentielle

— *Applications* : trier des cartes perforées ; trier des dates ; trier des chiffres + Exemple (Figure 2)

— *Propriétés* : tri stable ; pas en place (tri par dénombrement)

— *Complexité* : $O(n)$

Tri par dénombrement, tri par paquet (tri suffixe), tri par base

4 Affaiblissement des hypothèses pour le tri

Au début de la leçon nous avons posé quelques hypothèses sur le tri que nous allons faire (l'ordre que nous allons utilisé et l'espace mémoire nécessaire pour stoker l'ensemble des clés). Dans cette partie, nous allons affaiblir certaine d'entre elles pour examiner les tris que nous pouvons alors obtenir.

4.1 Tri topologique [2, p.565]

Hypothèse : on n'utilise maintenant plus qu'un ordre partiel : on utilise l'ordre donné par un graphe.

— Problème du tri topologique

entrée $G = (V, E)$ un graphe orienté acyclique.
sortie L une liste constituée des éléments de S telle que si $(u, v) \in E$, u apparaît avant v dans L.

— *Remarque* : ordre partiel induit par le graphe

— Algorithme du tri topologique et correction DEV

4.2 Tri externe

On s'autorise des données qui ne tiennent pas en mémoire vive.

Ouverture

Tri de shell
Réseau de tri

Quelques notions importantes

Les tris par comparaisons

Algorithmes naïfs On commence par traiter des algorithmes de tri naïf. Ce sont des algorithmes qui n'utilise aucun paradigmes ni aucune structures de données élaborées.

Tri par sélection Le tri par sélection [3, p.310] recherche le minimum parmi les éléments non triés pour le placer à la suite des éléments déjà triés. Lorsque l'on recherche séquentiellement le minimum et qu'on l'échange avec le premier élément non trié, nous réalisons une sélection ordinaire (Algorithme 1).

Afin de simplifier les calculs de complexité, nous allons donner l'algorithme itératif équivalent (en terme de correction et de complexité) de cette sélection ordinaire (Algorithme 2).

Algorithm 1 Algorithme récursif du tri par sélection classique.

```
1:  function TRI-SÉLECTION(A, i)   ▷ A : tab à trier ; i ∈ ℕ
2:      if i < n then
3:          j ← i
4:          for k = i + 1 à n do                ▷ Recherche séquentielle du min
5:              if A[k] < A[j] then
6:                  j ← k
7:              end if
8:          end for
9:          Échanger A[i] et A[j]               ▷ Placement du min
10:         Tri-Sélection(A, i + 1)             ▷ Tri fin tableau
11:     end if
12: end function
```

Algorithm 2 Algorithme itératif du tri par sélection classique.

```
1:  function TRI-SÉLECTION-ITER(A)
2:      i ← 1
3:      while i < n do
4:          j ← i
5:          for k = i + 1 à n do
6:              if A[k] < A[j] then
7:                  j ← k
8:              end if
9:          end for
10:         Échanger A[i] et A[j]
11:         i ← i + 1
12:     end while
13: end function
```

Théorème (Complexité)**.** *Le tri par sélection ordinaire s'exécute en* $\Theta(n^2)$ *(en moyenne et dans le pire cas).*

Démonstration. Pour analyser la complexité de cet algorithme, nous allons analyser le nombre de comparaisons effectué ainsi que le nombre d'échange lors du tri.

Pour toute liste de taille n, on effectue $n-1$ comparaisons pour trouver le minimum. Cette recherche est ensuite suivie de la même recherche sur une liste à $n-1$ éléments. En notant $Max_C(n)$ et $Moy_C(n)$ le nombre maximal (moyen) de comparaison pour une liste à n éléments, on trouve : $Max_C(n) = n - 1 + Max_C(n-1)$ pour $n > 1$ et $Max_C(1) = 0$. Comme, le nombre de comparaison ne dépend pas de la liste : $Moy_C(n) = Max_C(n)$. L'équation de récurrence sur Max se résout selon une méthode directe, on obtient : $Max_C = \frac{n(n-1)}{2}$. Le nombre de comparaisons que l'on effectue est donc $\Theta(n)$.

Le nombre d'échange dans le pire cas est le même qu'en moyenne car on ne fait qu'un échange lors de l'appel du tri. On a donc $Max_E = Moy_E = n - 1 = \Theta(n)$. □

Une autre implémentation d'un tri par sélection est un tri à bulle (Algorithme 3). Son principe, joliment présenté par son nom, consiste à faire remonter les plus petit éléments en tête du tableau (comme des bulles). Pour cela, on part de la fin du tableau et tant que l'élément est plus petit que les autres on effectue une permutation.

Algorithm 3 Algorithme du tri par dénombrement.

```
1: function TRI-BULLE(A)                                   ▷ A : tableau à trier
2:     i ← 1
3:     while i < n do
4:         for j = n à i + 1 do
5:             if A[1] < A[j − 1] then
6:                 Échanger A[j − 1] et A[j]
7:             end if
8:         end for
9:         i ← i + 1
10:    end while
11: end function
```

Théorème (Complexité)**.** *Le tri à bulle s'exécute en* $\Theta(n^2)$ *(en moyenne et dans le pire cas).*

Démonstration. Pour analyser la complexité de cet algorithme, nous allons analyser le nombre de comparaisons effectué ainsi que le nombre d'échange lors du tri.

Le nombre de comparaison est exactement le même que dans un tri par sélection ordinaire : $\Theta(n^2)$.

Le nombre d'échange dans le pire cas n'est plus le même qu'en moyenne car on le nombre d'échange lors de l'appel du tri varie en fonction de la place de l'élément. On utilise deux méthodes pour calculer ce nombre d'échange : un calcul par dénombrement ou un calcul par séries génératrices. Dans les deux cas, on trouve une complexité moyenne en $O(n^2)$. □

Tri par insertion Le tri par insertion [3, p.320] consiste à pré-trier une liste afin d'entrer les éléments à leur bon emplacement dans la liste triée. Par exemple à l'itération i, on insère le $i^{\text{ième}}$ élément à la bonne place dans la liste des $i-1$ éléments qui le précède (cette liste est triée par construction de l'algorithme). Comme pour le tri par sélection, il existe plusieurs tri par insertion : le tri par insertion séquentiel ou le tri par insertion dichotomique.

Le tri par insertion séquentiel (Algorithme 4) effectue la recherche de la place de l'élément à insérer séquentiellement : on parcours toute la liste au pire cas. La fonction récursive décrivant le tri par insertion que l'on donne n'est pas récursive terminal et peut difficilement l'être. Le tri ainsi défini n'est pas en place. Cependant, on peut donné un algorithme itératif qui conserve le nombre de comparaison ainsi que le nombre d'échange et qui rend le tri en place. On a alors bien un tri en place avec une complexité en $O(n^2)$.

Algorithm 4 Algorithme récursif du tri par insertion séquentiel.

```
1:  function TRI-INSERTION(A, i)        ▷ A : tab à trier ; i ∈ ℕ
2:      if i > 1 then
3:          Tri-Insertion(A, i − 1)          ▷ Trier début tab
4:          k ← i − 1                        ▷ Recherche rang de i
5:          x ← A[i]
6:          while A[k] > x do
7:              A[k + 1] ← A[k]
8:              k ← k − 1
9:          end while
10:         A[k + 1] ← x                     ▷ On place i
11:     end if
12: end function
```

Algorithm 5 Algorithme itératif du tri par insertion séquentiel.

```
1:  function TRI-INSERTION-ITER(A)
2:      for k = 2 à n do
3:          k ← i − 1
4:          x ← A[i]
5:          while A[k] > x do
6:              A[k + 1] ← A[k]
7:              k ← k − 1
8:          end while
9:          A[k + 1] ← x
10:     end for
11: end function
```

Proposition. *Le tri par insertion a une complexité en $O(n^2)$ dans le pire des cas et en moyenne.*

Démonstration. Nous allons commencer par l'analyse du nombre de comparaisons dans le pire cas. Dans ce cas, on doit faire i comparaisons après l'appel de Tri-Insertion(t, $i-1$) (on le fait pour tous les appels lorsque le tableau est trié dans l'ordre décroissant). On obtient alors la relation de récurrence suivante : $Max_C(n) = Max_C(n-1) + n$ pour $n > 1$ et $Max_C(1) = 0$. On a alors $Max_C(n) = \frac{n(n+1)}{2} - 1$. Donc le nombre de comparaisons dans le pire cas est $O(n^2)$.

Pour la complexité moyenne, on utilise les résultats établis pour l'analyse du tri à bulle... □

Tri sous le paradigme diviser pour régner Le paradigme diviser pour régner est utile lors de la définition d'un tri. Comme l'ordre est total nous pouvons séparer en deux sous-ensemble les données à trier afin d'en simplifier le problème. Les deux algorithmes qui existent à ce sujet choisissent de mettre la difficulté à deux endroits différents : le tri rapide découpe l'ensemble de manière complexe afin de les rassembler très facilement (au pire c'est une concaténation de listes) ; le tri fusion découpe simplement l'ensemble mais demande une petite astuce lors de la fusion. Nous allons étudier ces deux méthodes, puis nous les comparerons.

Tri rapide Le tri rapide (Algorithme 6) est un tri en place dont la complexité moyenne est en $O(n \log n)$ (elle est optimale) et dont la complexité au pire cas est en $O(n^2)$. Il applique le paradigme diviser pour régner en séparant l'entrée en deux sous tableau dont les valeurs sont respectivement inférieures (ou supérieures) à un pivot. L'algorithme de tri se rappelle alors sur ces deux sous tableau.

Algorithm 6 Algorithme du tri rapide.

```
1: function TRI-RAPIDE(A)  ▷ A : tableau à trier
2:     if A.taille ≥ 2 then
3:         pivot ← A[0]
4:         i ← 0
5:         j ← A.taille − 1
6:         while i < j do
7:             if A[i + 1] ≥ pivot then
8:                 Échanger A[i + 1] et A[j]
9:                 j ← j − 1
10:            else
11:                Échanger A[i + 1] et A[i]
12:                i ← i + 1
13:            end if
14:        end while
15:        Tri-Rapide(A[0 . . . i − 1])
16:        Tri-Rapide(A[i + 1 . . . A.taille − 1])
17:    end if
18: end function
```

Algorithm 7 Algorithme du tri rapide randomise.

```
1: function TRI-RAPIDE(A)  ▷ A : tableau à trier
2:     if A.taille ≥ 2 then
3:         pivor ← Random(A) ▷ On prend un élément au hasard dans A
4:         i ← 0
5:         j ← A.taille − 1
6:         while i < j do
7:             if A[i + 1] ≥ pivot then
8:                 Échanger A[i + 1] et A[j]
9:                 j ← j − 1
10:            else
11:                Échanger A[i + 1] et A[i]
12:                i ← i + 1
13:            end if
14:        end while
15:        Tri-Rapide(A[0 . . . i − 1])
16:        Tri-Rapide(A[i + 1 . . . A.taille − 1])
17:    end if
18: end function
```

Théorème (Correction)**.** *L'algorithme du tri rapide (Algorithme 6) est correct.*

Démonstration. content... □

Analyse de la complexité du tri rapide

— Dans le pire cas :

— Dans le cas favorable

Remarque. L'équilibre du découpage du tableau en deux sous tableau se répercute dans la complexité d'exécution.

Tri rapide randomisé Pour étudier la complexité moyenne de ce tri, nous allons utiliser une version randomisé qui simplifiera notre étude (Algorithme 7). On remarque que la correction du nouvel algorithme ne change pas car elle ne dépend pas du pivot choisi mais uniquement des actions autour de celui-ci.

Tri fusion Le tri fusion n'est pas un tri en place mais un tri optimal dans le pire cas et en moyenne. Il se base également sur le paradigme diviser pour régner : il coupe le tableau en deux sous tableau qu'il tri séparément, puis il les réassemble dans une opération de combinaison. Contrairement au tri rapide, c'est cette dernière qui est la plus complexe à réaliser.

Algorithm 8 Algorithme de fusion dans le tri fusion [1, p.129].

```
1:  function FUSION(A, g, m, d)                ▷ A : tab à trier
2:      for i = g à m do
3:          R[i] ← A[i]
4:      end for
5:      for j = m + 1 à d do
6:          R[j] ← A[d + m + 1 − j]
7:      end for
8:      i ← g; j ← d
9:      for k = g à d do
10:         if R[i] ≤ R[j] then
11:             A[k] ← L[i]
12:             i ← i + 1
13:         else
14:             A[k] ← R[j]
15:             j ← j + 1
16:         end if
17:     end for
18: end function
```

Algorithm 9 Algorithme du tri fusion.

```
1:  function TRI-FUSION(A, i, j)
2:      if i < j then
3:          n ← j − i + 1
4:          g ← i
5:          m ← i + ⌊n/2⌋ − 1
6:          d ← j
7:          Tri-Fusion(A, g, m)
8:          Tri-Fusion(A, m + 1, d)
9:          Fusion(A, g, m, d)
10:     end if
11: end function
```

Théorème (Correction)**.** *La fonction Fusion du tri fusion (Algorithme 8) est correct.*

Démonstration. On exhibe un invariant pour la boucle POUR de la fonction Fusion : à chaque itération k de la boucle, le tableau $A[g \dots k-1]$ contient les $k-g$ plus petits éléments de R, les sous tableaux $R[g \dots m]$ et $R[m+1 \dots d]$ sont triés et $R[i]$ et $R[j]$ sont les plus petits éléments de ces sous-tableau. On prouve cet invariant par récurrence sur le nombre de boucles.

Initialisation Avant la première itération, $k = g$, le sous tableau $A[g \dots k-1]$ est vide et donc contient les $k - g = 0$ plus petits éléments de R. De plus, $i = g$ et $j = m+1$ donc par hypothèse sur le tri des sous tableau de R, $R[i]$ et $R[j]$ sont les plus petits éléments de ces sous-tableau.

Conservation Pour montrer que la boucle conserve son invariant, supposons que $R[i] \leq R[j]$. Alors, $R[i]$ est le plus petit élément qui n'a pas été copié dans A. Comme, $A[g \dots k-1]$ contient les $k-g$ plus petits éléments de R, $A[g \dots k]$ contient les $k-g+1$ plus petits éléments de R lorsqu'on copie $R[i]$. De plus, l'incrémentation de i assure que $R[i]$ et $R[j]$ sont les plus petits éléments de ces sous-tableau.

Terminaison A la fin de la boucle $k = d+1$, tous les éléments du tableau R ont été recopié dans le tableau A qui contient les $d+1-g$ plus petites valeurs triées.

□

Remarque. On remarque que le temps d'exécution de l'algorithme Fusion est en $\Theta(n)$.

Théorème (Complexité)**.** *Le tri fusion (Algorithme 9) s'exécute en $O(n \log n)$ dans le pire cas.*

Démonstration. Nous allons appliquer le master theorem. Pour cela, on suppose que n est une puissance de deux (l'algorithme marche de même si ce n'est pas le cas, mais l'application du master theorem qui nous donne le même résultat est plus délicate). Nous souhaitons alors mettre en place la récurrence $T(n)$ dans le pire cas.

Diviser Cette étape calcul le milieu du tableau, nous avons un coût constant : $\Theta(1)$

Régner On résout deux problèmes récursivement de taille $\frac{n}{2} \in \mathbb{N}$ (par hypothèse sur n), nous avons un temps d'exécution en $2T(\frac{n}{2})$.

Combiner Par la remarque précédente, on a $C(n) = \Theta(n)$

On obtient alors

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ 2T(\frac{n}{2}) + cn & \text{si } n > 1 \end{cases}$$

Le master theorem, nous donne alors la complexité dans le pire cas suivante : $\Theta(n \log n)$. □

Optimalité du tri par comparaison

Les tris linéaires

Tri par dénombrement Le tri par dénombrement [2, p.180] suppose qu'il existe un entier k (connu à l'avance) tel que l'ensemble des données d'entrée soit contenu dans l'intervalle 0 à k. Le tri est linéaire si $k = O(n)$. Pour effectuer le tri on cherche à savoir combien d'éléments sont inférieurs à i pour tout $i \leq k$. Nous pouvons alors placer l'élément i à la bonne place dans le tableau de sortie (si 4 est précédé de 15 éléments, on le place à la 16$^{\text{ième}}$ place). On utilise alors un espace de stockage temporaire : : un table de taille k afin que chacune de ces cellules contiennent le nombre d'éléments inférieurs ou égaux à celle-ci (Algorithme 10). Nous donnons un exemple de l'exécution de cet algorithme (Figure 1)

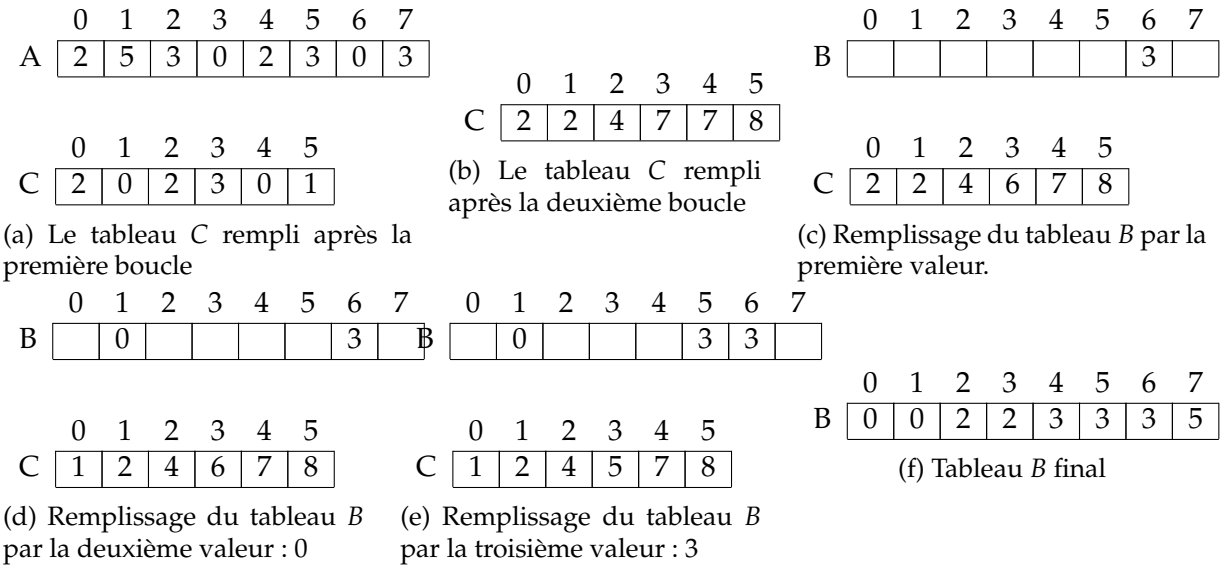
Algorithm 10 Algorithme du tri par dénombrement.

```
1:  function TRI-DÉNOMBREMENT(A, B, k)        ▷ A : tableau à trier ; B : tableau trié ; k : borne
2:      C un nouveau tableau de taille k initialisé à 0
3:      for j = 0 à A.longueur − 1 do          ▷ Combien d'élément sont égaux à i dans C[i].
4:          C[A[j]] ← C[A[j]] + 1
5:      end for
6:      for i = 1 à k do                       ▷ Combien d'élément sont inférieurs ou égaux à i dans C[i].
7:          C[i] ← C[i] + C[i − 1]
8:      end for
9:      for j = A.longueur − 1 à 0 do          ▷ Tri des éléments dans B
10:         B[C[A[j]]] ← A[j]
11:         C[A[j]] ← C[A[j]] − 1               ▷ Gestion des valeurs multiples
12:     end for
13: end function
```

Complexité : L'analyse des trois boucles FOR successives nous donne une complexité en $O(n + k)$. On ne retrouve pas la borne pour les tris par comparaison car à la place de comparaison, on utilise les indexe d'un tableau pour trier les éléments.

Le tri par dénombrement n'est pas en place (il faut stocker le tableau C). Cependant, le tri par dénombrement est un tri stable. La stabilité en elle même n'est pas nécessairement une propriété fondamentale mais comme le tri par dénombrement est une sous-routine du tri par base, sa stabilité devient très importante. La stabilité provient de la décroissance dans la dernière boucle de l'algorithme.

Tri par base Le tri par base [2, p.182] permet de trié des clés constitué de d sous-clés entières bornées de poids plus ou moins fort. Une idée naïve serait de commencer par trier les sous-clés de poids les plus forts en premier. Même si cette méthode fonctionne, il nous faudrait stocker



(a) Le tableau C rempli après la première boucle

(b) Le tableau C rempli après la deuxième boucle

(c) Remplissage du tableau B par la première valeur.

(d) Remplissage du tableau B par la deuxième valeur : 0

(e) Remplissage du tableau B par la troisième valeur : 3

(f) Tableau B final

FIGURE 1 – Tri-Dénombrement sur un tableau contenant huit éléments dont la borne est cinq.

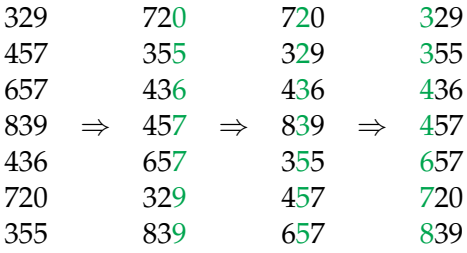


FIGURE 2 – Algorithme du tri par base sur une liste de nombres à trois chiffres.

des piles de clés (déduite par le tri que l'on vient de faire). En pratique, le tri par base se fait en commençant par la clé de poids minimale. Cette sous-routine de tri se fait à l'aide d'un tri par dénombrement car sa stabilité est importante pour garder un ordre cohérent lors du tri.

Applications : trier des cartes perforées (ce sont les premiers supports de programmes) qui possède 80 sous-clés sur les caractères alpha-numériques ; trier des dates (ou plus généralement des données qui s'étalent sur plusieurs champs) ; trier des entiers (ou plus généralement des suites de bits)

Lemme. *Étant donné n nombres de d chiffres dans lesquels chaque chiffre peut prendre k valeurs possible, le tri par base se fait en $\Theta(d(n+k))$ si le tri stable employé comme sous-routine se fait en $\Theta(n+k)$.*

Démonstration. Par récurrence sur le nombre de colonne. □

Quand d est constant et que $k = O(n)$, le tri par base se fait en temps linéaire. On dispose alors d'une certaine souplesse dans la décomposition de nos clés.

Tri par paquets Le tri par paquet suppose que l'entrée provient d'une distribution uniforme (on suppose que l'on a un processus aléatoire qui distribue les éléments de manière uniforme et indépendante sur l'intervalle $[0,1[$) [2, p.185]. Comme précédemment, nous avons un tri rapide grâce aux hypothèses sur les entrées (ici la distribution uniforme).

Le tri par paquets (Algorithme 11) commence par diviser notre intervalle en n sous-intervalles, $\left[\frac{k}{n}, \frac{k+1}{n}\right[$ (qui l'on nomme paquets) de même taille. Ensuite nous plaçons les différentes entrées dans leur paquets correspondants pour les trier à l'aide d'un tri par insertion. On parcours alors les paquets enfin d'en énumérer leur contenu (trié).

Algorithm 11 Algorithme du tri par paquets.

```
1:  function TRI-PAQUETS(A)                          ▷ A : tableau à trier
2:      n ← A.longueur
3:      B un nouveau tableau de taille n initialisé à la liste vide
4:      for i = 0 à n − 1 do                         ▷ Répartir l'entrée dans les paquets
5:          Insérer A[i] dans la liste B[⌊nA[i]⌋]
6:      end for
7:      for i = 0 à n − 1 do                         ▷ Tri les paquets
8:          Trier la liste B[i] à l'aide d'un tri par insertion
9:      end for
10:     Concaténer les listes B[0] à B[n − 1]
11: end function
```

Théorème (Correction)**.** *L'algorithme Tri-Paquets est correct.*

Démonstration. Soit deux éléments $A[i]$ et $A[j]$ tels que $A[i] < A[j]$ (sans perte de généralité). On a alors $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$. On distingue deux cas : soit $A[i]$ et $A[j]$ sont envoyés dans le même paquets ; soit $A[i]$ et $A[j]$ sont envoyé dans deux paquets différents avec le paquet de $A[i]$ devant celui de $A[j]$. Si $A[i]$ et $A[j]$ sont envoyés dans le même paquets alors le tri par insertion place $A[i]$ devant $A[j]$. Sinon, la concaténation des listes nous assure que $A[i]$ est devant $A[j]$. □

Théorème (Complexité)**.** *La complexité espérée de l'algorithme Tri-Paquets est en $O(n)$.*

Démonstration. Pour analyser le temps d'exécution, on remarque que toutes les opérations (sauf le tri par insertion) se fond en $O(n)$ dans le pire cas. Analysons maintenant le résultat de tous ces tris par insertion. Notons n_i la variable aléatoire qui désigne le nombre d'éléments placés dans le paquet $B[i]$. Comme le tri par insertion s'effectue en temps quadratique, le temps d'exécution du tri par paquet est

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Nous analysons maintenant le temps moyen en calculant l'espérance du temps d'exécution, l'espérance étant prise sur la distribution en entrée. En prenant l'espérance de notre identité et par sa linéarité, on obtient :

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \textstyle\sum_{i=0}^{n-1} O(n_i^2)\right] && \text{(passage à l'espérance)} \\ &= \Theta(n) + \textstyle\sum_{i=0}^{n-1} E\left[O(n_i^2)\right] && \text{(linéarité de l'espérance)} \\ &= \Theta(n) + \textstyle\sum_{i=0}^{n-1} O(E\left[n_i^2\right]) && \text{(linéarité espérance + } O \text{ donne multiplication constante)} \end{aligned}$$

Montrons maintenant que $E\left[n_i^2\right] = 2 - \frac{1}{n}$ pour tout $i \in \{0, \dots n-1\}$. Il n'y a rien d'étonnant que toutes les valeurs des $E\left[n_i^2\right]$ soit égales car les valeurs d'entrée sont identiquement distribuées. On définit les variables aléatoires indicatrice $X_{ij} = I\{A[j] \text{ va dans le paquet } i\}$ pour tout $i \in \{0, \dots, n-1\}$ et $j \in \{1, \dots, n\}$. Par conséquent, $n_i = \sum_{j=1}^{n} X_{ij}$ (c'est la probabilité que l'entrée j tombe dans le paquet i et ce pour toutes les entrées). Pour calculer $E[n_i^2]$, on développe le carré et on regroupe les termes, on obtient :

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=1}^{n-1} X_{ij}\right)^2\right] \\ &= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\ &= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{i\le j\le n}\sum_{\substack{1\le k\le n\\ k\neq n}} X_{ij}X_{ik}\right] \\ &= \sum_{j=1}^{n} E\left[X_{ij}^2\right] + \sum_{i\le j\le n}\sum_{\substack{1\le k\le n\\ k\neq n}} E\left[X_{ij}X_{ik}\right] \end{aligned}$$

Comme $X_{ij} = 1$ avec une probabilité de $\frac{1}{n}$ et 0 sinon, on a

$$\begin{aligned} E[X_{ij}^2] &= 1^2\tfrac{1}{n} + 0^2\left(1-\tfrac{1}{n}\right) \\ &= \tfrac{1}{n} \end{aligned}$$

De plus, quand $j \neq k$, les variables X_{ik} et X_{ij} sont indépendantes, d'où :

$$\begin{aligned} E[X_{ij}X_{ik}] &= E[X_{ij}]E[X_{ik}] \\ &= \tfrac{1}{n}\tfrac{1}{n} \\ &= \tfrac{1}{n^2} \end{aligned}$$

D'où en substituant dans l'équation, on a

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^{n}\tfrac{1}{n} + \sum_{i\le j\le n}\sum_{\substack{1\le k\le n\\ k\neq n}}\tfrac{1}{n^2} \\ &= n\tfrac{1}{n} + n(n-1)\tfrac{1}{n^2} \\ &= 1 + \tfrac{n-1}{n} \\ &= 2 - \tfrac{1}{n} \end{aligned}$$

En utilisant cette espérance, on conclut que la complexité du tri par paquets, en moyenne, est $\Theta(n) + nO(2-\frac{1}{n}) = \Theta(n)$. □

Remarque. Même si l'entrée ne provient pas d'une distribution uniforme, le tri par paquet peut encore être en temps linéaire si la somme des carrés des tailles des paquets est une fonction linéaire en le nombre d'objets.

Références

[1] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique*. Manuels informatiques Masson. Masson, 1992.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique, 3ème édition*. Dunod, 2010.

[3] C. Froidevaux, M.C. Gaudel, and M. Soria. *Types de données et algorithmes*. McGraw-Hill, 1990.