Étude et présentation d'un algorithme

Polytech Sorbonne

Algorithmique et structures de données

Algorithme de tri Radix-Sort

Auteur :
Jordy Ajanohoun

Enseignante :
Cécile Braunstein

4 juin 2018

Table des matières

1 Problème algorithmique du tri

On veut trier une liste d'éléments qui est dans le désordre, ou plus précisément dans un ordre qui ne nous convient pas. On désigne par **tri** l'opération consistant à **ordonner** un ensemble d'éléments en fonction de **clés** sur lesquelles est définie une **relation d'ordre**. L'objectif du tri, en tant qu'algorithme, est de mettre les éléments dans le bon ordre. Les algorithmes de tri ont une grande importance pratique. Ils sont fondamentaux dans certains domaines, comme l'informatique de gestion où l'on tri de manière quasi-systématique des données avant de les utiliser. De plus, ils sont **utiles à de nombreux algorithmes** plus complexes dont certains algorithmes de recherche, comme la recherche par dichotomie. Des études tendent d'ailleurs à montrer qu'environ **un quart des cycles machine** sont utilisés pour trier. L'étude du tri est également intéressante en elle-même car il s'agit sans doute de l'un des domaines de l'algorithmique qui a été le plus étudié et qui a conduit à des résultats remarquables sur la construction d'algorithmes et l'étude de leur complexité.

Historiquement, le dépouillement des recensements aux USA a été l'une des premières tâches automatisées grâce à une machine. Répondant à un concours du gouvernement américain, **Herman Hollerith** réalisa la première machine à statistiques en **1890**. Cette machine trie automatiquement des cartes perforées. Par la suite, de grandes compagnies comme IBM proposèrent plusieurs modèles comme par exemple le modèle D3 qui pouvait trier 1000 cartes par minute. La plupart de ces machines utilisent la méthode du **tri radix** (Radix-Sort), car elle permet d'ordonner efficacement les cartes perforées.

C'est sur cette méthode que se porte mon étude. Néanmoins, il est important de rappeler qu'il existe une multitude d'algorithmes de tri parmi lesquels le tri fusion, tri rapide, tri à bulles... Quand on tri une petite liste et qu'on le fait rarement, même le plus mauvais des algorithmes fera l'affaire. Par contre, quand on manipule des listes conséquences et/ou qu'on les trie souvent, il faut **choisir avec attention** celui qui sera le plus **adapté au contexte**.

2 À chaque problème sa solution : le tri radix sort

L'algorithme de **tri par base** (Radix-Sort en anglais) date de l'époque des **cartes perforées**. On utilise ce tri lorsque les éléments à trier sont des **chaînes de caractères et/ou des nombres** car il permet de trier uniquement dans **un ordre lexicographique**. Les clés sont donc les éléments à trier et l'**unique clé de tri** est l'ordre lexicographique. Il appartient à la famille des algorithmes de tri utilisant la structure des données car il **tri sans effectuer une seule comparaison**. Ce n'est pas un algorithme de tri par comparaison.

2.1 Principe du Radix Sort

Il y a en fait principalement 2 façons d'effectuer le tri radix : **LSD** et **MSD**. Cependant, le principe de base est exactement le même.

Algorithme Radix-Sort (LSD)

1. choisir une base : 2, 10 ou hexadécimal
2. choisir k $\in \mathbb{N}^*$
3. prendre les k premiers digits les moins significatifs de chaque clé (en fonction de la base choisie)
4. trier les clés selon ces k digits seulement, mais conserver l'ordre des clés ayant les mêmes k digits (ce qui est un **tri stable**)
5. répéter le tri avec chaque groupe de k digits plus significatif. On tri donc par groupe de digits, des moins significatifs aux plus significatifs.

LSD pour "Least Significant Digit" car en effet on commence par trier les éléments en fonction de la valeur de leurs digits les moins significatifs. Contrairement au Radix-Sort MSD "Most Significant Digit" où on trie des digits les plus significatifs aux moins significatifs. C'est essentiellement l'étape 3 qui change, le point de départ. On trie les clés en fonctions de la valeur de leurs digits, position par position, de la gauche vers le droite pour le MSD.

Il faut préciser ce qu'on appelle "**digit**" ici et ce qu'on veut dire par "groupe de digits". En **base 10**, rien de plus simple ! Un digit est tout simplement un chiffre et un groupe de digits forment un nombre. En **binaire**, un digit correspond à un bit et un groupe de k digits est un mot/nombre binaire sur k bits. Enfin, en **hexadécimal** c'est similaire. Par exemple, 0_xFD3A est un groupe de 4 digits en base hexadécimal.

La notion de tri stable a été mentionnée, il est temps d'en dire un peu plus sur cette propriété qu'ont certains algorithmes de tri. On dit qu'un algorithme de tri est stable s'il ne modifie pas l'ordre initial des clés identiques. Un exemple concret s'impose pour illustrer l'algorithme et tout ce qui vient d'être dit.

On veut trier le tableau : 13 5 972 42 272 25 77 36

1. On choisit la base 10 pour un exemple simple et rapide
2. On prend $k = 1$ pour les mêmes raisons
3. On considère le premier chiffre le moins significatif de chaque nombre

   13 5 972 42 272 25 77 36
4. On trie les clés en fonction de ce chiffre en conservant l'ordre des éléments qui ont la même valeur pour ce chiffre

   972 42 272 13 5 25 36 77
5. On répète le tri avec chaque chiffre plus significatif

   Après la 2^e itération : 5 13 25 36 42 972 272 77
   Après la 3^e itération : 5 13 25 36 42 77 272 972

Le tableau est trié ! Néanmoins, plusieurs remarques sont à faire.

Premièrement, comment trier les **chaînes de caractères** avec cet algorithme ? C'est simple, il faut se rappeler que chaque caractère correspond à un **code ASCII** codé sur 1 octet. A partir de là il y a un ordre lexicographique et une correspondance à des nombres qu'on peut ainsi trier. Cela requière cependant d'être plutôt à l'aise avec la manipulation de bits et d'octets dans un langage qui le permet. En effet, dans l'algorithme explicité plus haut, **un "digit"** correspond alors à **un octet** dans le cas ou on souhaite trier des chaînes de caractères. En reprenant l'algorithme et en remplaçant le mot "digit" par "octet", on se rend vite compte des difficultés pour l'implémenter.

Deuxièmement, on voit que **les clés peuvent être de différentes tailles** alors combien d'itérations faut-il faire ? Combien de fois il faut répéter l'étape 5 pour que la liste soit triée ? Par **taille de clé** j'entends le nombre de digits qui la compose. Par exemple, la clé 0_d113 est de taille 3 et 0_xFD3A de taille 4. La taille des clés est à exprimer **en fonction de la base choisie**. En effet, 0_x1010 est de taille 4 alors que son équivalent en décimal 0_d10 a une taille égale à 2. Cette notion de taille de clé a son importance puisqu'elle permet de **déterminer le nombre d'itérations** à faire. En effet, il va dépendre de la taille de la clé la plus grande car il faut s'assurer que l'ensemble des digits significatifs de toutes les clés soit traité. Dans notre exemple de tri, la plus grande clé est 972 et sa taille est 3, nous avons pris $k = 1$ donc il nous a fallu 3 itérations. Si on appelle m la plus grande taille de clé, i**l faudra au plus** $\frac{m}{k} + 1$ **itérations** pour trier la liste (division entière). C'est logique car **à chaque itération on traite k digits** de chaque clé et il y a **au maximum m digits** d'après nos notations.

Dernière remarque, pour les clés dont la **taille** est **inférieure à** m. Il faut considérer que toutes les clés sont de même taille m en mettant à 0 les digits manquants à gauche (les plus significatifs) de manière à ce que la valeur de ces clés reste inchangée. $0_d005 = 0_d5$! C'est ce qui permet à chaque itération de bien traiter **l'ensemble** des clés (étape 5 de l'algorithme). Sans cette considération, pour l'implémentation il faut gérer le fait que les itérations doivent prendre fin plus tôt sur les clés les plus petites tout en continuant sur les plus grandes. C'est contraignant et plus complexe pour au final trier en un nombre total d'itérations qui est **le même dans les deux cas**. C'est uniquement le travail fait à chaque itération qui diminuera car on aura exclu au fur et a mesure les clés dont on a traité tous les digits. En revanche, il y aura du travail supplémentaire pour gérer les itérations et maintenir un résultat correcte. D'où la considération.

2.2 Mon implémentation

J'ai choisi de l'implémenter en **langage C** car c'est le langage le plus **bas niveau** dans lequel je possède le plus d'expérience. Je me suis orienté vers un langage bas niveau notamment pour la manipulation de bits et octets et la vitesse. En effet, les itérations de l'algorithme en base 2 (binaire) se font sur les bits, on traite les clés bit par bit pour les trier.

Lorsqu'on cherche à implémenter ce tri radix, on se rend vite compte du problème suivant : à l'étape 4 comment trier **sans comparaison** ? Il faut avoir accès aux digits à traiter et pouvoir réarranger les clés en fonction de ces digits mais sans faire de comparaisons ! D'où l'utilisation des **structures de données**. Il y a plusieurs moyens de gérer et d'implémenter cela. Pour ma part, j'ai choisi d'implémenter le radix sort de **2 manières différentes** : une première utilisant les **tableaux** et une autre faisant appelle à la structure de donnée **FIFO** !

Avant tout on a besoin de connaître la plus grande taille de clé de la liste à trier sinon on ne peut pas avoir de condition d'arrêt. C'est le but de ma fonction **getTailleClefInt** qui ne sera pas détaillée ici car ce n'est pas le sujet. Se référer à mon code source pour plus d'informations et précisions la concernant.

Je ne présenterai ici que le radix sort utilisant la **structure de donnée FIFO** car c'est la **plus efficace** des deux à mes yeux. Nous allons reprendre le contexte de la liste que nous cherchions à trier plus haut (**base 10,** $k = 1$**)**. Cela correspond à ma fonction **rdxSortPosIntFile_De** et ce tri radix **LSD** en pseudo code est le suivant :

Data: Tab : Tableau d'**entiers positifs** de **taille n**
Result: Le tableau Tab trié (le tableau d'origine est modifié)
begin
  TAILLE_CLE_MAX ← getTailleClefInt(Tab, n) ;
  Initialisation de 10 files d'entiers (**FIFO**) vides indexées de 0 à 9 ;
  for *i allant de 1 à TAILLE_CLE_MAX* **do**
    for *each key in Tab* **do**
      u ← le i-ème chiffre significatif de la clé en partant de la droite ;
      Insérer la clé dans la file d'indice u (add key to Queue u) ;
    end
    j ← 0 ;
    for *v allant de* 0 *à* 9 **do**
      while *Queue v is not empty* **do**
        Tab[j] = dequeue Queue v (pop la file v) ;
        j ← $j + 1$;
      end
    end
  end
end

Algorithm 1: rdxSortPosIntFile_De — Algorithme de mon implémentation du tri Radix-Sort LSD utilisant la structure de donnée FIFO.

En exécutant cet algorithme sur notre exemple ou un autre, on obtient exactement le tableau trié ! On peut très facilement adapter cet algorithme à d'autres bases et pour des $k \geq 2$. Il suffit d'adapter le nombre de files qui dépend de la base et de k et de modifier la première boucle pour ne plus traiter chiffre par chiffre mais bit par bit par exemple en base 2 avec $k = 1$. Dans ce cas, il y aura seulement 2 files : une pour les clés dont le ième bit vaut 0 et l'autre 1. C'est justement une des versions supplémentaires que j'ai implémentée : **rdxSortPosIntFile_Bi** (Cf. **rdxSort.h** et **rdxSort.c** pour plus de détails). Le nombre de files correspond au nombre d'informations différentes qu'on peut "coder" avec k digits de la base choisie. A noter que la structure **FIFO** nous permet d'avoir un **tri stable**.

Pourquoi la structure de donnée FIFO ?

Une alternative simple pour ne pas utiliser de files consiste à utiliser des **tableaux** à la place. C'est exactement le même principe à la seule différence qu'on remplace "file" par "tableau". Mais cette seule différence entraîne un lourd **inconvénient**. En réfléchissant quelques minutes, on s'aperçoit vite que si on veut refaire la même chose avec des tableaux, il nous faut renseigner **la taille** de ces derniers. Or **comment savoir à l'avance** combien de nombres dans la liste auront le ième chiffre significatif égal à 1 ou 2 ou 3 ainsi de suite. C'est gérable sur une petite liste si on connaît tous les nombres à l'avance mais dès lors que la liste contient plus de 40 nombres à trier par exemple, cela devient très ardu voir impossible. En effet, même si on connaît l'état des chiffres significatifs à traiter à l'itération i, il est fortement susceptible d'être différent à l'itération i+1. En résumé, c'est **inenvisageable** d'allouer à chaque itération des tableaux avec des tailles qui correspondent exactement à nos besoins. Il n'y a pas d'autres choix que d'**allouer tableaux de taille** n pour être **garanti d'avoir suffisamment de place**. Cela veut dire **10 tableaux de taille n**, c'est conséquent en termes d'occupation mémoire lorsque n tend vers l'infini. J'ai néanmoins également implémenté cette version utilisant les tableaux (fonction **rdxSortPosIntTab_Bi**).

Ce n'est désormais plus un mystère, la structure de donnée FIFO représente un **gain de mémoire** dans notre cas. On alloue de la mémoire **uniquement lorsqu'on en a besoin**, sur commande, pas de surplus contrairement aux tableaux. Ceci est dû au fait qu'avec les files, ce n'est que lorsqu'on souhaite y ajouter un élément que l'on va consommer de l'espace mémoire pour stocker cet élément dans la structure. Tandis qu'avec les tableaux on va réserver $10 \times n$ emplacements mémoire pour n'en utiliser que n au total.

Qu'en est-il des entiers négatifs ?

Jusqu'à présent l'ensemble de ce qui a été dit est valable lorsqu'on veut trier des **nombres entiers positifs** ! Si vous avez été attentif, vous avez dû remarquer qu'il n y a pas encore eu l'apparition d'un seul entier négatif. Nous n'en n'avons pas encore parlé et c'est le moment. L'algorithme **ne fonctionnera pas** sur un tableau d'entiers comprenant des **entiers positifs et des entiers négatifs**. La raison est simple, lorsqu'on défile les files une à une jusqu'à ce qu'elles soient toutes vides on commence par la file des nombres dont le chiffre traité vaut 0 puis on va vers 9. Logique puisque $0 < 9$ mais dans le cas des entiers négatifs **il faut prendre en**

compte le signe et $0 > -9$. Cela devient alors complexe de gérer simultanément les positifs et les négatifs d'une même liste pendant le tri. La meilleure solution que j'ai trouvée consiste à **regrouper les positifs dans un sous tableau et les négatifs dans un autre sous tableau**, trier les 2 distinctement toujours par le principe du radix sort et enfin **fusionner** les 2 sous tableaux triés. Le résultat final est alors le **tableau d'origine** contenant des nombres positifs ET négatifs **trié**.

L'algorithme pour trier le sous tableau d'entiers négatifs est le même que pour les entiers positifs sauf que l'on défile les files en commençant par la file d'indice 9 et on va vers celle d'indice 0.

Il faut garder à l'esprit que nous nous sommes essentiellement concentrés sur l'algorithme en base 10 avec $k = 1$ mais qu'on peut tout à fait et simplement modifier ce dernier pour d'autres bases et k supérieur ! Peu importe la base, plus k est grand, plus on peut coder d'informations donc plus il faudra de files pour chacune d'entre elles. On peut donc adapter le radix sort donné ici à ses besoins et à son cas aisément avec un brin d'astuce.

2.3 Preuve de correction

Maintenant prouvons que l'algorithme se termine et donne le bon résultat. Nous avons au total 4 boucles dans le pseudo code. Trois boucles sont imbriquées dans une 4^e plus grande qui les englobe toutes. Nous appellerons cette 4^e boucle la boucle principale. Commençons dans l'ordre par **la première boucle que l'on rencontre dans la boucle principale** :

Preuve La preuve de terminaison de cette boucle est trivial. C'est un simple parcours du tableau Tab qui est de taille n finie. On insert chaque clé dans une seule file en fonction de son ième chiffre significatif ! Ni plus ni moins. Le nombre de files est également fini. L'insertion dans la bonne file se fait en un temps constant de même que l'accès au ième digit d'un nombre. **Aucune instruction de cette boucle ne peut durer un temps infini** si elle est implémentée sans faute. **Quand on sort de cette boucle les clés se trouvent dans les bonnes files, il ne reste plus qu'à les réarranger dans le tableau**.

La 2^e boucle interne que l'on rencontre dans la principale (numérotée 3) contient une autre boucle. C'est **la boucle la plus interne de l'algorithme**.

Preuve La preuve de terminaison de cette dernière est également trivial. Pop un élément d'une file se fait en un temps constant, de même que avoir accès à une case du tableau dont on connaît déjà l'indice. Il y a au plus n éléments dans les files, toutes files confondues et n est finie. Ainsi, si les files sont bien implémentées, **la file d'indice v mettra nécessairement un temps fini à se vider dans le tableau**. De plus, l'indice j pour remplir le tableau Tab avec le nouvel arrangement est **toujours mis à jour**. On écrase ni ne perd aucune donnée.

Revenons à la **boucle numérotée 3** :

Preuve v va de 0 à 9 soit 10 répétitions. Ces 10 répétitions correspondent aux 10 files indexées de 0 à 9. On ne fait rien d'autre que vider la file v dans le tableau Tab et nous avons déjà prouvé la terminaison et la correction de cette étape. **Cette**

boucle aussi ne peut pas durer infiniment. Avant cette boucle, **j est toujours réinitialisé à 0**. Ainsi, d'après les preuves précédentes, à la terminaison de cette boucle le tableau Tab contient toujours les n éléments de départ mais dans un ordre différent. **Si on ne regarde que les ièmes chiffres les moins significatifs des clés, ils sont dans l'ordre croissant** !

Pour finir il nous reste à prouver la terminaison et la correction de **la boucle principale** :

Invariant A chaque fin de passage dans la boucle, si on ne tient compte que des nombres que forment les i chiffres les moins significatifs des clés, le tableau est trié.

Init Pour $i = 1$ c'est vrai d'après les 3 précédentes preuves. L'initialisation est vérifiée.

Récurrence Si on ne tient compte que des nombres que forment les i chiffres les moins significatifs des clés, le tableau est trié. D'après les preuves précédentes, après le i+1 ème passage dans la boucle, les i+1 èmes chiffres les moins significatifs des clés sont désormais dans l'ordre croissant aussi ! Donc maintenant, si on ne regarde que les i+1 chiffres les moins significatifs des clés, ils forment un tableau de nombre trié. L'invariant est vérifié pour $i' = i + 1$.

Final La boucle va de $i = 1$ à $i = TAILLE\_CLE\_MAX = m$ étant **le nombre de chiffres composant la plus grande clé en valeur absolue**. Ainsi, pour $i = m$ l'ensemble du tableau est trié, l'algorithme donne donc le bon résultat.

Terminaison Boucle for (trivial)

L'algorithme est correct ! CQFD.

2.4 Analyse de la complexité

Analysons la complexité en temps de l'implémentation.

```
begin
    TAILLE_CLE_MAX ← getTailleClefInt(Tab, n) <————————Θ(n + m) ;
    Initialisation de 10 files d'entiers (FIFO) vides indexées de 0 à 9 <– Θ(1) ;
    for i allant de 1 à TAILLE_CLE_MAX do
        for each key in Tab do
            u ← le i-ème chiffre significatif de la clé <————————————Θ(1) ;
            Insérer la clé dans la file d'indice u (add key to Queue u) <—Θ(1) ;
        end
        j ← 0 ;
        for v allant de 0 à 9 do
            while Queue v is not empty do
                Tab[ j ] = dequeue Queue v ( pop la file v) <————————Θ(1) ;
                j ← j + 1 ;
            end
        end
    end
end
```

La fonction **getTailleClefInt parcours le tableau de taille n** une fois entièrement pour identifier la clé la plus grande en valeur absolue $\Theta(n)$. Ensuite, elle fait des **divisions successives** par puissances de 10 pour **compter le nombre de chiffres** qui composent cette clé donc au total m divisions $\Theta(m)$. Ces deux étapes ne sont pas imbriquées, elles se font l'une après l'autre donc la **complexité en temps** est $\Theta(n+m)$.

Complexité en temps de la boucle principale (la plus externe) :

$$T(n,m) = \sum_{i=1}^{m}(n \times 2 \times \Theta(1) + n \times \Theta(1)) = \sum_{i=1}^{m}(3 \times n \times \Theta(1))$$
$$\Leftrightarrow T(n,m) = \Theta(n) \times m = \Theta(n \times m)$$

Complexité en temps de l'algorithme Radix-Sort implémenté :

$$T(n) = \Theta(n \times m) + \Theta(n+m) = \Theta(n \times m)$$

Les opérations pop et enfile (insertion) de la structure de donnée file sont en $\Theta(1)$ car on a un accès direct à la tête et à la queue de la file par définition d'une file. A chaque passage dans la boucle principale, on pop n fois et on enfile (insert) n fois également. On passe m fois dans la boucle principale !

La complexité en occupation mémoire est $\Theta(n)$ comme je l'ai déjà expliqué, grâce à l'utilisation de la structure FIFO. On utilise uniquement l'espace mémoire dont on a besoin. On copie les clés du tableau dans une autre structure le temps des les réarranger dans un ordre plus lexicographique dans le tableau à chaque itération.

Il faut savoir qu'il existe d'autres versions du Radix-Sort, des variantes utilisant des comparaisons et même un autre algorithme de tri. C'est l'un des algorithmes de tri les plus rapides mais il nécessite assez d'assomptions et on ne peut trier que dans un ordre lexicographique avec.

Merci pour votre lecture !

Pour compiler : gcc -o exe rdxSort.c queue.c mainRdxSort.c -lm

Sources :

1. http://desaintar.free.fr/tipe/algo_tri.pdf
2. http://www.giacomazzi.fr/infor/Tri/TRadix.htm
3. http://www.irem.univ-bpclermont.fr/IMG/pdf/2FicheScientifique-3.pdf
4. https://haltode.fr/algo/tri/tri_base.html