ALGORITHMIQUE 1

TP1 : Algorithmes de Tri

Juba Ouarab

Elsbeth Monrroy

Licence 2 Informatique

- / 02 / 2021

Les algorithmes qui suivent furent réalisés conformément aux explications données dans le polycopié du cours, ainsi que dans le livre de référence « Introduction à l’algorithmique » de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

1. **Algorithme de Tri à Bulle**

Le tri à bulle est un des algorithmes de tri les plus simples qui consiste à parcourir une fois la liste pour chaque élément afin de pouvoir le placer à l’endroit qui lui correspond.

Voici le pseudo-code :

**TRI\_A\_BULLES:**

**Entrée : Un tableau S de taille n  
Sortie : Le tableau S trié en ordre croissant**

**Début :**

**Pour (i de n-1 à 0) {**

**Pour (j de 0 à i -1) {**

**Si (S[j] > S [j + 1]) {**

**PERMUTER (S; j ; j + 1) ;**

**}**

**}**

**}**

**Retourner S ;**

**Fin**

La complexité de cet algorithme, selon les notations de Landau correspond à O(n2)

Le temps d’exécution moyen pour une entrée de taille 1000 est de : 5 840 318.0 nanosecondes

1. **Algorithme de Tri Fusion**

Comme il a été expliqué en cours, le tri fusion a l’avantage d’appliquer la méthode de diviser pour régner afin d’optimiser son temps d’exécution. Ce tri va décomposer le tableau en deux sous tableaux, et les trier puis fusionner récursivement.

**TRI\_FUSION:**

**Entrée : Un tableau S de taille n  
Sortie : Le tableau S trié en ordre croissant**

**Début :**

**Si (longueur(S) > 1) {**

**DÉCOMPOSER (S, S1, S2) ;**

**S1 = TRI\_FUSION(S1) ;**

**S2 = TRI\_FUSION(S2) ;**

**S = FUSIONNER (S1, S2) ;**

**}**

**Retourner S ;**

**Fin**

La fusion de deux tableaux se réalise en parcourant chaque élément des tableaux, et en fonction de leurs valeurs, les ranger dans un tableau auxiliaire, ici appelé « fusion ». Pour cela on fait croitre l’indice de parcours soit d’un tableau soit de l’autre. A la fin, on rajoute tout élément manquant, et on retourne le résultat de la fusion.

**FUSIONNER :**

**Entrée : Deux tableaux S1 et S2 de tailles respectives n1 et n2**

**Sortie : Un tableau « fusion » contenant les éléments de S1 et S2 triés**

**Début :**

**index1= 0 ;**

**index2= 0 ;**

**Tant que (index1 < n1 et index2 < n2) {**

**Si (S1[index1] <= S2[index2]) {**

**RAJOUTER (fusion, S1[index1]);**

**index1++;**

**Sinon {**

**RAJOUTER (fusion, S2[index2]);**

**index2++;**

**}**

**}**

**Si (index1 < n1) {**

**RAJOUTER (fusion, S1[index1, n1-1]);**

**Sinon {**

**RAJOUTER (fusion, S2[index2, n2-1]);**

**}**

**Retourner fusion;**

**Fin**

Cet algorithme récursif admet une complexité donnée par l’équation récursive :

*T(n) = 2n + 1 + 2T(n/2)*

Ceci revient donc à une complexité de l’ordre de Θ(n log n).

Le type d’implémentation choisi pour cet algorithme a un impact considérable dans le temps d’exécution.

Une première implémentation fut faite à l’aide des Linked List, ou on peut accéder aux éléments de la liste en faisant appel à la méthode get (int index). Cependant, cette implémentation est inefficace pour des entrées de l’ordre de 105.

Pour donner un exemple, le temps d’exécution moyen pour une entrée de taille 1000, avec une liste est de 5 251 572.5 nanosecondes. Le temps d’exécution de l’algorithme de tri fusion est presque égale à celui du tri à bulle ; pourtant, sa complexité est largement inférieure à n2.

Ce manque d’efficacité nous a poussé à chercher une meilleure implémentation pour ce tri. Ainsi, on a finalement choisi de se servir de la structure Queue. En accédant aux éléments de la queue avec les méthodes peek() et poll(), on a pu réduire le temps d’exécution à

1 593 136.4 nanosecondes, soit une différence de 3 658 436.1.

1. **Algorithme de Tri Rapide avec la partition dite du drapeau**
2. **Algorithme de Tri par Tas**

Cet algorithme repose sur la structure de tas et donc d’arbre binaire tassé. Pour l’implémentation de cette structure, ainsi que du tri correspondant, on s’est servi des explications données dans le livre de référence. Puisqu’on cherche trier un tableau en utilisant les tas, on ne va implémenter que les fonctions nécessaires à la construction d’un tas, et à la préservation des propriétés des tas.

On sait qu’un tas peut être représenté par un tableau, si on considère qu’à chaque nœud correspond un élément du tableau. Pour vérifier qu’un tableau représente correctement un tas (et si nécessaire, pour le modifier), on considère l’algorithme suivant:

**ENTASSER :**

**Entrée : Un tableau A de taille n et un indice i**

**Début :**

**g = GAUCHE(i) ; //indice de la racine du sous arbre gauche du**

**nœud A[i]**

**d = DROITE(i) ; // indice de la racine du sous arbre droit du**

**nœud A[i]**

**Si (g<=n et A[g] > A[i]) {**

**max= g ;**

**Sinon**

**max = i ;**

**}**

**Si (d<=n et A[d]>A[max]) {**

**Max = d ;**

**}**

**Si (max ! = i) {**

**PERMUTER (A[i], A[max]) ;**

**ENTASSER (A, max) ;**

**}**

**Fin**

Cette fonction s’exécute en O(log n)

L’algorithme va déplacer l’élément d’indice i de telle sorte que toute la structure de tas soit préservée. Pour cela elle va chercher l’élément maximal parmi l’élément i et ses deux sous arbres, si nécessaire faire des permutations afin que l’élément maximal se trouve à la racine, puis répéter la procédure dans tous les sous arbres qui auraient pu être affectés par cette permutation.

Remarque : La longueur du tableau est le nombre d’éléments du tableau, alors que la taille du tableau est le nombre d’éléments du tas rangés dans le tableau.

Une fois qu’on a un outil qui nous permet de valider un tas, on peut passer à la construction ce celui-ci. Cette procédure est très simple, elle consiste à entasser tous les éléments d’un tableau donné, sauf ceux qui correspondent à des feuilles du tas, puisque celles-là sont déjà des tas de taille 1. Du fait qu’on représente le tas par un tableau à n éléments, on constate que les feuilles correspondent aux éléments indexés par , + 1, + 2, …, n-1. La construction d’un tas à partir d’un tableau se fait donc de la façon suivante :

**CONSTRUCTION :**

**Entrée : Un tableau A de longueur n**

**Début :**

**taille[A] = n ;**

**Pour (i de -1** **à 0){**

**ENTASSER(A, i) ;**

**}**

**Fin**

Cette fonction s’exécute en O(n)

Pour trier le tableau, il suffit de constater que grâce à la structure de tas, l’élément maximal va toujours se trouver au sommet de l’arbre. Autrement dit, dans un tableau représentant un tas, l’élément maximal va se trouver à la position 0. En sachant ceci, on pourra directement le placer à l’endroit qui lui correspond, donc à l’indice n-1. Cependant, même si cet élément est correctement placé, la permutation réalisée peut avoir des conséquences dans le reste du tas. Ainsi, on devra à nouveau entasser tous les éléments du tableau, sauf ceux qui sont correctement placés. Voici l’algorithme qui décrit cette procédure :

**TRI\_PAR\_TAS :**

**Entrée : Un tableau A d’entiers de longueur n, et de taille m  
Début :**

**CONSTRUCTION(A) ;**

**Pour (i de n-1 à 1) {**

**PERMUTER(A[0], A[i]) ;**

**m -- ;**

**ENTASSER (A, 0) ;**

**}**

**Fin**

La complexité de cet algorithme est de l’ordre de (n log n), il est donc un algorithme très efficace et peu couteux. Ceci est clairement reflété par le temps moyen d’exécution : 502 296.03 nanosecondes pour un tableau de taille 1000.

1. **Tri par Base**

Le tri par base est un tri sans comparaison qui se base sur un principe d’indexation. Il fait appel au tri par dénombrement pour chacun des chiffres des chacun des nombres contenus dans le tableau d’entrée.

Voyons premièrement le tri par dénombrement.

Ce tri prend en entrée un tableau contenant des entiers entre 0 et 9 inclus, et va les trier en fonction des indices qui leur correspondent. Dans un premier temps, on initialise un tableau rempli de zéros, et dont la taille est l’élément maximal du tableau d’entrée. On le rempli en enregistrant le nombre d’apparitions de chaque élément du tableau d’entrée. En sachant le nombre d’occurrence de chaque chiffre, on peut connaitre l’espace du tableau que ces répétitions vont occuper, et donc aussi l’indice du chiffre suivant. Ainsi on déclare un tableau qui calcule cet indice en fonction de l’indice précédent et du nombre d’occurrences. Finalement, en connaissant tous les indices, on remplit un tableau avec les éléments du tableau d’entrée, à la position donnée par le tableau d’indices.

Voici le pseudo-code décrivant cette procédure :

**TRI\_PAR\_DENOMBREMENT :**

**Entrée : Un tableau T à n éléments (tous compris entre 0 et 9), et dont l’élément maximal est k.**

**Sortie : Un tableau R trié, contenant les éléments de T.**

**Début :**

**Pour (i de 0 à k-1){**

**nb[i] = 0 ;**

**}**

**Pour (i de 1 à n) {**

**nb[T[i]] = nb[T[i]] + 1 ;**

**}**

**pos[0] = 0 ;**

**Pour (i de 1 à k-1) {**

**pos[i] = pos[i-1] + nb[i-1];**

**}**

**Pour (i de 1 à n) {**

**R[pos[T[i]]] = T[i] ;**

**pos[T[i]] = pos[T[i]] + 1 ;**

**}**

**Retourner R**

**Fin**

Le tri par base s’appuie sur ce principe de tri par indexation, mais en rangeant des nombres entiers en fonction de leurs chiffres. Si « c » est le nombre de chiffres de l’élément maximal du tableau, il va donc falloir faire appel à la fonction TRI\_PAR\_DENOMBREMENT « c » fois, et trier en fonction du chiffre d’indice 1, puis 2, 3, et ainsi de suite jusqu’à c. Ceci est possible en paramétrant la fonction, et en créant une fonction auxiliaire qui va extraire d’un nombre le chiffre à l’indice donné :

**EXTRAIRE\_CHIFFRE :**

**Entrée : un nombre num et un indice index**

**Sortie : le chiffre de num à l’indice index.**

**Début :**

**Retourner RESTE(num, 10^index) ;**

**Fin**

La fonction RESTE(a, b) retourne le reste de la division euclidienne de a par b.

Finalement, le tri par base fait appel à la version paramétrée TRI\_PAR\_DENOMBREMENT(Tableau T, int indice) du tri par dénombrement :

**TRI\_PAR\_BASE :**

**Entrée : Un tableau T dont l’élément maximal a n chiffres.**

**Sortie : Un tableau trié**

**Début :**

**Pour (i de 1 à n){**

**T = TRI\_PAR\_DENOMBREMENT(T, i) ;**

**}**

**Retourner T ;**

**Fin**

Cet algorithme à une complexité linéaire : En effet, pour chaque chiffre de chaque nombre on fera appel au tri par dénombrement, qui prend un temps de O(n + k), ce qui donne alors une complexité de O(c(n+k)).

Le temps moyen d'exécution pour un tableau à 1000 éléments est : 2 687 440.8 nanosecondes, mais avec des très grands écarts qui vont de 4 133 200 à 19 480 600 nanosecondes.