Compte-rendu d'Algorithmique

Comparaison des méthodes de tri

Introduction

Un algorithme de tri peut avoir de nombreux intérêts pour manipuler des éléments d'un ensemble, il peut être utile, en préalable à un traitement de données, sans oublier son rôle principal qui est de trier ces éléments, considérons par exemple un problème de recherche d'un élément dans une collection, dans ce cas, il peut être intéressant de commencer par trier ces données, pour faciliter après la recherche au terme de complexité, car on aura juste à utiliser la méthode de recherche dichotomique qui aura un coût logarithmique.

Le but de ce projet est d'étudier les trois méthodes de tri à savoir le tri à bulles, le tri rapide,et finalement le tri par dénombrement. Certains tris sont plus efficaces que d'autres suivant le contexte d'utilisation, nous avons dû mettre en place dans les travaux pratiques deux classes «TabElements» et «Elements» pour pouvoir ensuite trier un tableau d'éléments quelconques. La classe «TabElements» est représentée par deux attributs un tableau de la classe «Elements» ainsi que sa taille. En revanche, la classe «Element» est représentée par une clé qui est de type entier et une valeur de type String. Vous trouverez dans le même fichier le code des tris implémentés en java dans la classe «TabElements», avec des commentaires qui explique le fonctionnement de chaque tri. Par ailleurs, pour désigner le nombre de comparaisons et le nombre d'affectations. Dans toute la suite de ce compte-rendu, on considère un tableau d'entiers de taille 5.

I.le tri à bulles

I.1.Principe

On parcours le tableau, par exemple de gauche à droite, en *comparant les éléments du tableau deux à deux* en les permutant s'ils ne sont pas dans le bon ordre. Au cours d'une passe du tableau, les plus grands éléments remontent vers la droite comme des **bulles vers la surface.**

I.2.Exemple : [6 3 5 1 0]

- Premier passage :

(6 3 5 1 0)	➔	(3 6 5 1 0)	3<6 échange
(3 6 5 1 0)	➔	(3 5 6 1 0)	5<6 échange
(3 5 6 1 0)	➔	(3 5 1 6 0)	1<6 échange
(3 5 1 6 0)	➔	(3 5 1 0 6)	0<6 échange

- Deuxième passage :

(3 5 1 0 6)	➔	(3 5 1 0 6)	Pas d'échange
(3 5 1 0 6)	➔	(3 1 5 0 6)	1<5 échange
(3 1 5 0 6)	➔	(3 1 0 5 6)	0<5 échange

- Troisième passage :

(3 1 0 5 6)	➔	(1 3 0 5 6)	1<3 échange
(1 3 0 5 6)	➔	(1 0 3 5 6)	0<3 échange

- Quatrième passage :

(1 0 3 5 6)	➔	(0 1 3 5 6)	0<1 échange

On obtient [0 1 3 5 6]

I.3.Complexité

```
Algorithme tri à bulles           //calcul de complexité
Variables : indiceFin et i (entiers), echange (booléen), tmp (Elements),
Entrée : tableau tabElement d'éléments non triés et sa taille Taille
sortie :  le tableau tabElement  trié
Début
        indiceFin ← Taille - 1     // affectation, soustraction:+2
        echange ← VRAI    // affectation: +1
        TANTQUE  echange ← VRAI  faire      //Boucles : indiceFin fois
                 echange ← FAUX    // affectation: +1
                 POUR i allant de 1 jusqu'à indiceFin faire   /*Boucles : indiceFin-1+1 fois,
                                                              affectation, comparaison, addition,affectation : +4*/
                         SI tabElements[i - 1] > tabElements[i] faire          //comparaison : +1
                                          tmp ← tabElements[i]       // affectation: +1
                                          tabElements[i] ← tabElements[i - 1] // affectation: +1
                                          tabElements[i - 1] ← tmp   // affectation: +1
                                          echange ← true             // affectation: +1
                         FINSI
                 FINPOUR
                 indiceFin ←  indiceFin - 1 // affectation, soustraction:+2
        FINTANTQUE
Fin
```

En réalisant les différentes opérations et en calculant la somme des opérations élémentaires, on trouve que $C_{moy} = O(9 \times taille^2 + 3 \times taille + 3) = O(taille^2)$. Autrement dit, sa complexité moyenne est de $O(n^2)$, c'est à dire une complexité dite quadratique où n est le nombre d'éléments à trier. La complexité dans les meilleurs des cas concerne quand le tableau est déjà trié dans l'ordre croissant et la complexité dans le pire des cas concerne quand le tableau est trié dans l'ordre décroissant. Pour le pire des cas, on va obtenir la même complexité $O(n^2)$. Cependant, dans les meilleurs des cas, puisque le tableau est déjà trié, on aura fait qu'une seule itération dans le « **TANTQUE** », ce qui fait une complexité linéaire O(n). En effet, on peut donc en conclure que cette algorithme de tri paraît inefficace sur un très grand nombre d'éléments à trier surtout quand l'on sait que les meilleurs algorithmes de tri ont une complexité logarithmique (quasi-linéaire) de $O(n\log_2(n))$.

II.Tri rapide

II.1.Principe

Le **tri rapide**, ou **quicksort**, fait partie des tris qui utilisent les comparaisons. Son principe suit un approche qu'on appelle "diviser pour régner" : on sélectionne un élément de l'ensemble « **le *pivot*** » avant de placer d'un coté les éléments inférieurs ou égaux à cet élément, et de l'autre côté ceux qui lui sont supérieurs. On utilise pour ça un partitionnement. On va appliquer par la suite à nouveau la même procédure sur les deux sous-ensembles, jusqu'à ce qu'on tombe sur un tableau d'un seul élément, on peut dire donc que le tri rapide est un algorithme de tri récursif. Il peut se résumer de la manière suivante : On partitionne l'ensemble en deux et on trie séparément les deux partitions.

II.2.Exemple [7 9 2 8 4]

On choisit le pivot, dans notre cas je choisis le premier élément, 7

7	9	4	8	2

On découpe le tableau en trois parties, une partie avec des éléments inférieurs au pivot (4 et 2), la partie contenant le pivot (7), et une partie avec les éléments supérieurs au pivot (9 et 8). On peut déjà dire qu'on a placé le pivot à sa place définitive dans le tableau, puisque les autres éléments sont soit supérieurs soit inférieurs à lui

4	2	**7**	9	8

On recommence en choisissant de nouveau un pivot pour chaque sous tableaux créés(le premier élément de chaque tableau).

4	2	**7**	**9**	8

Dernière étape du partitionnement, désormais aucuns sous tableaux ne contient plus d'un élément, le tri est donc terminé.

2	**4**	**7**	8	**9**

II.3.Complexité

```
Algorithme  tri rapide

Entrée : tableau tabElement d'éléments non triés, l'indice de debut(entier)et l'indice de fin (entier)
sortie :  le tableau tabElement  trié
debut
     SI deb < fin          // comparaison:+1
         positionPivot ← partition(T, deb, fin)       //affectation : +1
         triRapide(T, deb, positionPivot - 1)         //appel récursif avec un coût = positionPivot-1-deb+1
         triRapide(T, positionPivot + 1, fin)         //appel récursif avec un coût = fin-positionPivot+1
     FINSI
Fin

SOUS-ALGORITHME partition( T : tableau d'Elements, deb : entier, fin : entier)
VARIABLE compt, i:entier ; pivot, tmp, : Elements
Début
    compt ← deb   // affectation :+1
    pivot ← T[deb]         // affectation :+1
    tmp ← new Elements()   // affectation :+1
    POUR i allant de deb+1 jusqu'a fin faire          //Boucle fin-(deb+1)+1 fois
        SI T[i] < pivot alors                         //comparaison : +1
            compt ← compt + 1                         // affectation,addition :+2
            tmp ← T[compt]                            // affectation :+1
            T[compt] ← T[i]                           // affectation :+1
            T[i] ← tmp                                // affectation :+1
         FINSI
    FINPOUR
    tmp ← T[deb]                                      // affectation :+1
    T[deb] ← T[compt]                                 // affectation :+1
    T[compt] ← tmp                                    // affectation :+1
    renvoyer (compt)                                  // affectation :+1
Fin
```

Pour calculer la complexité ce tri rapide, on va d'abord commencer par le calcul de complexité du sous-algorithme partition, on obtiendra donc on additionnant les opérations élémentaires un $C_{moy} = O(6 \times taille + 7) = O(taille)$. La complexité de l'algorithme de partitionnement sur le tableau de taille n est donc $\Theta(n)$. Concentrons nous maintenant sur l'algorithme tri rapide. Au premier appel

sur le tableau, l'algorithme contient deux appels récursifs, le premier traite la sous-tableau gauche le second la sous-tableau droite qui ont respectivement un coût T(positionPivot-deb) et T(fin-positionPivot+1). D'autre part, nous venons de voir que le partitionnement avait un coût $\Theta(n)$. L'expression récurrente de la complexité du tri rapide est donc

$$C_{moy}(T) = T(\text{positionPivot-deb}) + T(\text{fin-positionPivot+1}) + \Theta(n)$$

la complexité dans le pire des cas sera dans le cas où la méthode partitionne sépare le tableau en deux sous-tableaux de tailles déséquilibrées , par exemple dans le cas où **PositionPivot = deb** ou encore **PositionPivot = Fin** et ainsi l'une des deux sous-tableau est réduite à un unique élément. Il suffit de remplacer, dans ce cas on aura une complexité quadratique de $O(n^2)$. Cependant, la complexité dans les meilleurs des cas sera dans le cas où la méthode partitionne sépare le tableau en deux sous-tableaux de tailles égales, ainsi on obtiendra un nombre de «branches» qui sera une puissance de 2. Ce qui fait qu'on pourra toujours utiliser le logarithme binaire lors de la division. Finalement, la complexité meilleur du tri par rapide est quasi-linéaire de $O(n \log_2 n)$.

III.Tri par dénombrement

III.1.Principe

Le principe du tri par dénombrement est simple, on parcourt le tableau et on compte le nombre de fois que chaque élément apparaît. Une fois qu'on a le tableau de comptage *E* (avec *E[i]* le nombre de fois où *i* apparaît dans le tableau), on peut le parcourir dans le sens croissant (pour un tri croissant), et placer dans le tableau trié *E[i]* fois l'élément *i* (avec *i* allant de l'élément minimum du tableau jusqu'à l'élément maximum).

III.2.Exemple [2 3 5 1 1]

Tableau initial				
2	3	5	1	1

La valeur maximum ici dans le tableau ci-dessus est 5. On va donc créer un tableau de comptage secondaire de taille 6 (pour avoir le 0 compris) :

Tableau secondaire : initialisation						
Index	0	1	2	3	4	5
Occurrence	0	0	0	0	0	0

Le but va être maintenant de rajouter +1 à chaque fois qu'on trouve une occurrence qui correspond à l'index. Notre tableau secondaire va finalement nous donner ceci :

Tableau secondaire : comptage						
Index	0	1	2	3	4	5

Occurrence	0	**2**	**1**	**1**	0	**1**

Pour qu'au final on se retrouve avec un tableau trié :

Tableau initial : trié				
1	1	2	3	5

III.3.Complexité

```
Algorithme  tri par Dénombrement

Entrée : tableau tabElement d'éléments non triés, l'indice de debut(entier)et l'indice de fin (entier)
sortie :  le tableau tabElement  trié
VARIABLE i, position, taillemax :entier ; tab : tableau d'Elements ; tableauCle : tableau d 'entiers
debut
    taillemax ← max_tableau()                                   // affectation :+1

    //Initialisation du tableau des clés

    POUR i allant de 0 jusqu'a taille du tableauCle  - 1 faire  //Boucle tableauCle.taille fois
        tableauCle[i] ← 0                                       // affectation :+1

    //Remplissage du tableau des clés à partir de tableau qu'on veut trier

    POUR i allant de 0 jusqu'a  taille du tabElements - 1  faire  //Boucle tableauElements.taille fois
        tableauCle[tabElements[i] ← tableauCle[tabElements[i]]] + 1      // affectation,addition :+2

    POUR i allant de 1 jusqu'a taille du tableauCle faire      //Boucle tableauCle.taille - 1 fois

        tableauCle[i] ← tableauCle[i] + tableauCle[i - 1]       // affectation,addition :+2


    //recopie de tableau final

    POUR i allant de 0 jusqu'a  taille du tabElements - 1 faire  //Boucle tableauElements.taille fois
        position ← tableauCle[tabElements[i]]                   // affectation :+1
        tab[position - 1] ← tabElements[i]                      // affectation :+1
        tableauCle[tabElements[i]]] ← tableauCle[tabElements[i]]] - 1    // affectation,addition :+2

    POUR i allant de 0 jusqu'a  taille du tabElements - 1 faire          //Boucle tableauElements.taille fois
         tab[i] ← tabElements[i]                                         // affectation :+1

Fin
```

dans le calcul de la complexité de notre algorithme, on va considérer que la complexité de la fonction max_tableau() vaut le coût d'une affectation, même si sa complexité est linéaire O(n). En réalisant les différentes opérations élémentaires dans notre algorithme, on trouve que sa complexité moyenne est de :

$$C_{moy}=O(8\times taille+3\times tableauCle.taille-1)=O(taille+tableauCle.taille)$$

c'est à dire que la complexité moyenne de ce tri est de O(n + m) soit une complexité en temps linéaire avec n est le nombre d'éléments à trier et m le cardinal des éléments à trier. On peut trouver encore que la complexité dans les meilleurs des cas et la complexité dans le pire des cas sont également de O(n + m), car l'algorithme devra quand même dénombrer toutes les valeurs possibles. Le tri par dénombrement limité uniquement aux entiers positifs.

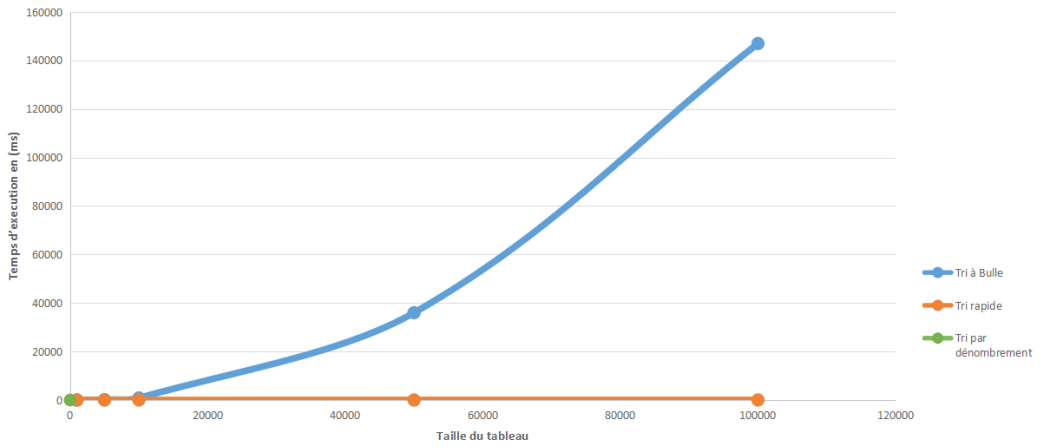
Comparaison entre les trois algorithmes des tri

Dans cette partie, on va essayer de comparer les trois au niveau de la complexité temporelle, pour cela on va procéder à la comparaison en temps d'exécution avec des tableaux des tailles différentes générées aléatoirement. Le temps donné va être en en millisecondes (ms) et vaut pour des clés allant de 0 à la taille du tableau. De plus, on ne prend pas en compte l'affichage du tableau, mais uniquement la fonction du tri qu'on souhaite comparer. On peut donc ranger les résultats sous forme d'un tableau :

Comparaison entre les trois algorithmes de tri					
La taille du tableau	1000	5000	10000	50000	100000
Tri à bulle	=37 ms	=173 ms	≈844 ms	≈36000 ms (36 secondes)	≈ 147000 (147 secondes)
Tri rapide	=1 ms	=4 ms	=10 ms	=24 ms	=40 ms
Tri par dénombrement	=1 ms	=4 ms	=9 ms	= 17ms	= 32 ms

On peut ainsi produire ce graphique avec les courbes suivantes :

Temps d'exécution des trois algorithmes de tri en fonction de la taille du tableau



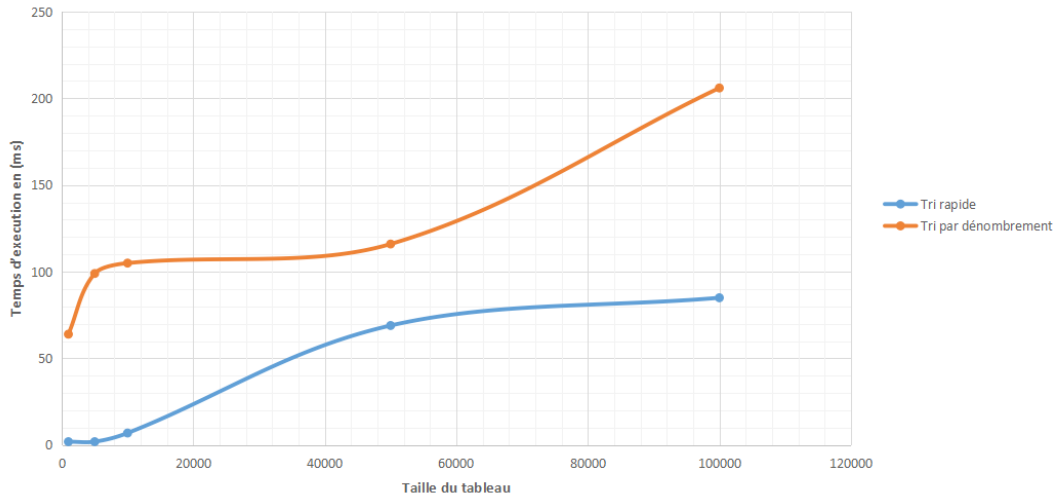
On constate bien que le tri par sélection est le tri le plus inefficace parmi les deux autres. Pour comparer de plus près le tri fusion et le tri dénombrement, les clés peuvent désormais prendre des valeurs comprises entre 0 et 100 millions. On range les résultats sous forme de tableau:

Comparaison entre les trois algorithmes de tri (avec des clés de 0 à 10 millions)					
La taille du tableau	1000	5000	10000	50000	100000
Tri par dénombrement	≈64 ms	≈99 ms	≈105 ms	≈119 ms	≈206 ms
Tri rapide	2 ms	2 ms	7 ms	≈69 ms	≈85 ms

On peut ainsi produire ce graphique avec les courbes suivantes :

Temps d'exécution des trois algorithmes de tri en fonction de la taille du tableau

(avec des clés de 0 à 10 millions)



Conclusion

Pour conclure, et en analysant les données résultat de la comparaison entre les trois tri, on peut dire ainsi que l'utilisation du tri à bulles est déconseillé du fait de sa complexité quadratique. Par contre, l'utilisation du tri par rapide (Quick Sort) et du tri dénombrement dépend du contexte où on utilise l'un des deux tri :

- Si par exemple on utilise dans notre tableau de clés connues come étant de petits nombres (moins de 1 millions) et étant aussi des entiers positifs, on utilisera dans ce cas le tri par dénombrement
- Si par contre les clés sont réelles ou sont des entiers suffisamment grands (plus de 10 millions) , alors le tri par dénombrement sera inutilisable avec les clés réelles et moins performant avec les grands nombres, et le tri rapide s'avérera être plus efficace vu sa complexité logarithmique

Néanmoins, la complexité du tri par dénombrement et du tri rapide étant similairement proche, il ne s'agira que d'une question de quelques millisecondes. On ne peut donc en réalité pas déterminer lequel des deux est le meilleur. En revanche le tri par dénombrement est restreint uniquement aux entiers positifs et limite encore les possibilités pour l'utiliser.
Ainsi, on ne peut pas affirmer sur l'efficacité d'un tri par rapport à l'autre, chaque tri a un intervalle d'utilisation avec lequel il est plus performant, bien que les tris avec une complexité linéaire ou quasi-linéaire sont plus performants.