Victor Vuillard – Thomas Petazzoni

Algorithmes de tris

Rapport Travaux Pratiques 1 – AG51

Semestre d'automne 2002-2003

**Université de Technologie
de Belfort Montbéliard**

Table des matières

Introduction

Les objectifs du TP étaient d'implémenter et de comparer le temps d'exécution de différentes méthodes de tri :

- Le tri par insertion qui caractérise une méthode de conception incrémentale.
- Le tri fusion qui utilise une méthode récursive.
- Le tri par tas basé sur une structure de données particulière.
- Le tri rapide qui adopte une stratégie dite "diviser pour règner".

Pour chacun de ces algorithmes, nous mesurerons un temps d'exécution de manière expérimentale et les comparerons avec les valeurs théoriques.

Notations

Dans l'ensemble du rapport, on notera :

- Tab un tableau
- n le nombre d'éléments du tableau à trier

1 Algorithmes et coûts des différents tris

1.1 Tri insertion

1.1.1 Algorithme

Le tri par insertion fonctionne de manière itérative. A chacune des itérations, une clé est comparée aux cases précédentes du tableau. Ces cases forment déjà un sous tableau trié, ne contenant au départ d'un seul élément. La clé est alors insérée à la bonne place du sous-tableau trié. Le sous-tableau trié croit jusqu'à atteindre la taille du tableau à trier.

Listing 1 – Tri par insertion

```
TriInsertion(Tab)
  pour j de 2 à longueur(Tab) faire
    clé <- Tab[j]
    i <- j - 1
    tant que (i > 0) et (Tab[i] > clé) faire
      Tab[i+1] <- Tab[i]
      i <- i - 1
    Fin tant que
    Tab[i+1] <- clé
  Fin pour
Fin
```

1.1.2 Coût théorique

Le coût de l'algorithme de tri par insertion est égal au coût de chacune de ses instructions multiplié par le nombre de fois que l'instruction est utilisée. Dans le cas de cet algo, chaque instruction à l'intérieur du `pour` est exécutée (n - 1) fois, sauf dans le `tant que` où elles seront exécutées le nombre de fois suivant :

$$\sum_{j=2}^{n}(t_j - 1) \tag{1}$$

Dans le cas le plus favorable, c'est à dire un tableau trié, les instructions à l'intérieur du `tant que` ne seront pas effectuées, car chaque élément testé sera inférieur ou égal à la clé. On aura alors un coût en $O(n)$. Dans le pire cas, où le tableau sera trié dans l'ordre inverse, les instructions de la boucle `tant que` seront exécutées à chaque passage. On aura alors une complexité en $O(n^2)$.

1.2 Tri fusion

1.2.1 Algorithme

Le tri par fusion utilise une méthode récursive. Il divise le tableau à trier en deux sous-tableaux qu'il fusionnera une fois triés. Le tri fusion est utilisé récursivement jusqu'à avoir des sous-tableaux à un seul élément.

En voici l'algorithme :

Listing 2 – Tri fusion

```
FUSION (Tab, p, q, r)
  n1 <- q - p + 1
  n2 <- r - q
  pour i de 1 à n1 faire
     L[i] <- Tab[p + i - 1]
  fin pour
  pour j de 1 à n2 faire
     R[j] <- Tab[q + j]
  fin pour
  L[n1 + 1] <- infini
  R[n2 + 1] <- infini
  i <- 1
  j <- 1
  pour k de p à r faire
     si (L[i] <= R[j]) alors
        Tab[k] <- L[i]
        i <- i + 1
     sinon
```

```
        Tab[k] <- R[j]
        j <- j + 1
      fin si
  fin pour
Fin


TRI-FUSION (Tab, p, r)
  si (p < r) alors
      q <- valeur_entiere((p + r) / 2)
      TRI-FUSION(Tab, p, q)
      TRI-FUSION(Tab, q+1, r)
      FUSION(Tab, p, q, r)
  fin si
Fin
```

1.2.2 Coût théorique

L'étape `Diviser` se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant, soit $O(1)$. Ensuite, l'étape `Régner` résout récursivement deux sous-tableaux chacuns de taille n/2, ce qui donne un temps en $2T(n/2)$. Enfin, l'étape `Combiner` correspond à la fusion sur un sous-tableau à n éléments et occupe un temps d'exécution en $O(n)$. L'exécution pocède autant d'occurences que de division du tableau, ce qui donne un $T(n) = O(n\ lg\ n)$.

1.3 Tri par tas

1.3.1 Algorithme

Un tas binaire est un tableau qui peut être vu comme un arbre binaire presque complet. Chaque noeud de l'arbre correspond à un élément du tableau qui contient la valeur de ce noeud. L'arbre est complètement rempli, sauf éventuellement au niveau le plus bas. Dans le tableau, on a alors :

Parent(i) Retourne l'indice : partie entière de i/2.

Gauche(i) Retourne l'indice : 2i.

Droite(i) Retourne l'indice : 2i + 1.

Un tas max est un tas pour lequel on a pour chaque noeud i autre que la racine :

$$Tab[Parent(i)] \geq Tab[i]. \tag{2}$$

Le tri par tas se base sur cette structure de tas max. Il commence d'ailleurs par en construire un à partir du tableau à trier. L'élément maximal est alors stocké dans la racine du tas, Tab[1]. Alors, on échange la racine avec le dernier élément du tableau et on enlève ce dernier élément du tas en réduisant la taille du tas de 1. On entasse le tas obtenu pour avoir à nouveau un tas max et on recommence l'opération jusqu'à avoir un tableau de seulement 2 éléments. Voici l'algorithme :

Listing 3 – Tri par tas

```
TriParTas(Tab)
  ConstruireTasMax(Tab)
  pour i de longueur(Tab) à 2 faire
     échanger Tab[i] et Tab[1]
      taille [A] <- taille[A] - 1
     EntasserMax(Tab, 1)
  fin pour
Fin

ConstruireTasMax(Tab)
  pour i de valeur_entiere (longueur(Tab) / 2) à 1 faire
     EntasserMax(Tab, i)
  fin pour
Fin

EntasserMax(Tab, i)
  l <- gauche(i)
  r <- droite(i)
  si (l <= taille(Tab)) et (Tab[l] >Tab[i]) alors
     max <- l
  sinon
     max <- i
  fin si
  si (r <= taille(Tab)) et (Tab[r] > Tab[Max]) alors
     max <- r
  fin si
  si (max != i) alors
     echanger Tab[i] et Tab[max]
     EntasserMax(Tab, max)
  fin si
fin
```

1.3.2 Coût théorique

La procédure TriParTas prend un temps $O(n\ lg\ n)$ puisque l'appel à ConstruireTasMax prend un temps en $O(n)$ et que chacun des n-1 appels à EntasserMax prend un temps $O(lg\ n)$.

1.4 Tri rapide

1.4.1 Algorithme

Comme le tri par fusion, le tri rapide est basé sur le paradigme `Diviser pour Régner`. Le tableau Tab[p..r] est d'abord partitionné en deux sous-tableaux Tab[p..q-1] et Tab[q+1..r] tels que chaque élément de Tab[p..q-1] soit inférieur ou égal à Tab[q] qui, lui-même, est inférieur ou égal à chaque élément de Tab[q+1..r]. L'indice q est calculé dans le cadre de cette procédure de partitionnement. Ensuite, les deux sous-tableaux Tab[p..q-1] et Tab[q+1..r] sont triés par des appels récursifs au tri rapide. L'avantage du tri rapide est que comme les sous-tableaux sont triés sur place, aucun travail n'est nécessaire pour les recombiner. Le tableau Tab[p..r] est immédiatement trié. Voici l'algorithme du tri rapide :

Listing 4 – Tri rapide

```
TriRapide (Tab, p, r)
  si (p < r) alors
     q <- Partition(Tab, p, r)
     TriRapide(Tab, p, q-1)
     TriRapide(Tab, q+1, r)
  fin si
Fin

Partition (Tab, p, r)
  x <- Tab[r]
  i <- p - 1
  pour j de p à r-1 faire
     si (Tab[j] <= x) alors
        i <- i + 1
        permuter Tab[i] et Tab[j]
     fin si
  fin pour
  permuter Tab[i+1] et Tab[r]
  retourner i+1
Fin
```

1.4.2 Coût théorique

Le temps d'exécution du tri rapide dépend du caractère équilibré ou non du partitionnement, qui dépend à son tour des éléments utilisés pour le partitionnement. Si le partitionnement est équilibré, l'algorithme s'exécute aussi rapidement que le tri fusion, soit en $O(n\ lg\ n)$. Par contre, si le partitionnement n'est pas équilibré, ce tri risque d'être aussi lent que le tri par insertion, soit en $O(n^2)$.

2 Relevés expérimentaux

2.1 Relevés

Chaque relevé a été effectué en réalisant une moyenne sur plusieurs itération de l'expérience dans le but d'obtenir des résultats aussi précis que possibles. D'autre part, pour chaque itération le même tableau est donné aux différents algorithmes de tri. Nous avons d'abord relevé des temps d'exécution dans le cas de tableau de faibles longueurs, dans le but d'observer les intervalles où tel ou tel tri est plus rapide.

Voici les résultats relevés, exprimés en millisecondes :

taille	*insertion*	*fusion*	*quicksort*	*tas*
0	0.0010	0.0030	0.0030	0.0020
20	0.0070	0.0060	0.0060	0.018
40	0.0050	0.012	0.012	0.012
60	0.0090	0.018	0.014	0.022
80	0.024	0.025	0.016	0.032
100	0.036	0.028	0.023	0.04
120	0.045	0.031	0.028	0.054
140	0.064	0.038	0.027	0.073
160	0.076	0.044	0.034	0.079
180	0.093	0.05	0.046	0.088
200	0.117	0.055	0.048	0.098
220	0.134	0.065	0.057	0.118
240	0.157	0.071	0.063	0.128
260	0.188	0.075	0.066	0.142
280	0.217	0.086	0.078	0.143
300	0.25	0.094	0.082	0.166
320	0.282	0.095	0.095	0.174
340	0.322	0.1	0.097	0.197
360	0.354	0.116	0.094	0.213
380	0.403	0.123	0.102	0.208
400	0.461	0.123	0.104	0.22
420	0.486	0.128	0.129	0.239
440	0.525	0.148	0.125	0.256
460	0.574	0.158	0.128	0.269
480	0.623	0.157	0.138	0.284
500	0.671	0.172	0.149	0.295

Pour mieux observer les tendances, nous avons ensuite réalisé de nouveau l'expérience sur des tableaux de plus grandes tailles.

Voici les résultats relevés, exprimés en millisecondes :

taille	*insertion*	*fusion*	*quicksort*	*tas*
2000	10.88	1.04	0.7	1.58
3000	23.74	1.14	1.18	2.2
4000	42.22	1.78	1.56	3.16
5000	66.0	2.08	2.04	4.04
6000	94.9	2.58	2.38	4.98
7000	129.2	3.02	2.94	5.9
8000	168.36	3.34	3.64	6.74
9000	213.44	4.02	3.94	7.76
10000	263.48	4.5	4.48	8.7
11000	318.54	5.02	4.96	9.74
12000	378.38	5.44	5.58	10.58
13000	445.36	5.88	6.0	11.72
14000	516.02	6.54	6.54	12.66
15000	591.56	7.1	6.98	13.76
16000	674.4	7.54	7.52	14.78
17000	760.14	8.08	8.08	15.84
18000	853.12	8.48	8.74	16.94
19000	950.64	9.14	9.2	17.94
20000	1053.52	9.62	9.88	19.0

Nous pourrons trouver en Annexe les courbes correspondant à ces différents relevés.

2.2 Analyse des résultats

Nous observons qu'effectivement les courbes suivent les tendances annoncées par la pratique. Le tri insertion suit un tendance en $ax^2 + bx + c$ avec un coefficient de régression de 0.988. On a donc bien une tendance en $O(n^2)$. Comme prévu, le tri rapide et le tri fusion qui ont tous deux un coût en $O(n\ lg\ n)$ sont quasiment confondus. La courbe pour le tri par tas, théoriquement en $O(lg\ n)$, reste bien proportionnelle à un coefficient près n aux courbes des deux autres tris en $O(n\ lg\ n)$. On observe également que le tri par insertion est très désavantageux pour les tableaux de grande taille. Pour des tableaux de 20 000 éléments, il est 100 fois plus lent qu'un tri fusion ou qu'un tri rapide. Toutefois, pour des tableaux de petites tailles, le tri insertion reste valable : en dessous de envirron 75 éléments, il trie plus vite que le tri rapide ou le tri fusion et il en est de même en dessous de 175 éléments si on le compare à notre implémentation du tri par tas.

3 Code source

Voici le code source de nos différents programmes de tri :

Listing 5 – Programmes de tri

```
import java.io.*;
import java.util.*;

public class tp1 {


  ////////////////////////////////
 //      Tri par insertion       //
////////////////////////////////

    static void triInsertion(int[] tableau) {

        int    j, cle;
        for (int i = 1; i < tableau.length; ++i) {
            cle = tableau[i];
            // On insere tableau[i] dans la sequence triee tableau[0..i-1]
            j = i;
            while (j > 0  && tableau[j-1] > cle) {
                tableau[j] = tableau[j-1];
                --j;
            }
            tableau[j] = cle;
        }
    }


  ////////////////////////
 //      Tri fusion      //
////////////////////////

    static void triFusion(int[] t) {
        int[] copie = (int[]) t.clone();
        triFusion(copie, t, 0, t.length);
    }

    static void triFusion(int source[], int destination[],
                          int debut, int fin) {
        int longueur = fin - debut;

        if (longueur <= 1) return;
        else {
            int milieu = (debut + fin)/2;
            triFusion(destination, source, debut, milieu);
```

```
            // source est trié de début à milieu
            triFusion(destination, source, milieu, fin);
            // source est trié de milieu à fin

            // Fusion des tableaux triés dans destination
            for (int i = debut, p = debut, q = milieu;
                 i < fin;
                 i++) {
                if (q>=fin || p<milieu &&
                    (source[p] < source[q])) {
                    destination[i] = source[p];
                    p++;
                } else {
                    destination[i] = source[q];
                    q++;
                }
            }
            // destination est trié de debut à fin
        }
    }


  ////////////////////////////
 //      Tri par tas       //
////////////////////////////



    static int gauche(int position)
    {
            return(2 * position);
    }

    static int droite(int position)
    {
            return(2 * position + 1);
    }

    static int parent(int position)
    {
            return((int)(position/2));
    }

    static void entasserMax(int []t, int pos, int taille)
    {
            int max;

            int g = gauche(pos);
            int d = droite(pos);
```

```
            if ((g <= taille) && (t[g] > t[pos]))
            {
                    max = g;
            }
            else
            {
                    max = pos;
            }
            if ((d <= taille) && (t[d] > t[max]))
            {
                    max = d;
            }
            if (max != pos)
            {
                    permute(t, pos, max);
                    entasserMax(t, max, taille);
            }
    }

    static void construireTasMax(int []t)
    {
            for (int i = (int)((t.length - 1) / 2); i >= 0; i--)
            {
                    entasserMax(t, i, t.length - 1);
            }
    }


    static void triTas(int [] t)
    {
        int taille = t.length - 1;

        construireTasMax(t);
        for (int i=t.length - 1; i > 0; i--)
        {
                permute(t, 0, i);
                 taille -= 1;
                entasserMax(t, 0, taille);
        }
    }


  ////////////////////////
 //      Tri rapide      //
////////////////////////

    static void QS_trier(int inf, int sup, int [] t)
    {
```

```
        int pivot;

        if (inf < sup)
        {
                pivot = QS_partition(t, inf, sup);
                QS_trier(inf, pivot - 1, t);
                QS_trier(pivot + 1, sup, t);
        }
    }

    static void permute(int []t, int a, int b)
    {
        int temp;

        temp = t[a];
        t[a] = t[b];
        t[b] = temp;
    }

    static int QS_partition(int [] t, int p, int r)
    {
        int x, i, j;

        x = t[r];
        i = p - 1;
        for (j = p; j < r; j++)
        {
                if (t[j] <= x)
                {
                        i += 1;
                        permute(t, i, j);
                }
        }
        permute(t, i+1, r);
        return(i+1);
    }

    static void triQuickSort(int [] t)
    {
        QS_trier(0, t.length - 1, t);
    }


  /////////////////////////
 // programme principal //
/////////////////////////

    public static void main(String[] ghg) {
```

```
Random rd = new Random();
int iteration = 50;
float temps_insertion = 0;
float temps_fusion    = 0;
float temps_qs    = 0;
float temps_tas    = 0;

System.out.println("Taille\tinsert\tfusion\tqs\ttas (ms)");

for (int n = 2000; n < 20001; n+=1000)
  {
    temps_insertion = 0;
    temps_fusion = 0;
    temps_qs = 0;
    temps_tas = 0;

    for (int k = 0; k < iteration ; k++)
      {
        int [] tab = new int[n];
        for (int i = 0; i < n; ++i) {
          tab[i] = rd.nextInt(10000);
        }
        int [] tab2 = (int[])tab.clone();
        int [] tab3 = (int[])tab.clone();
        int [] tab4 = (int[])tab.clone();

        Date d1 = new Date();
        triFusion(tab);
        Date d2 = new Date();
        triInsertion (tab2);
        Date d3 = new Date();
        triQuickSort(tab3);
        Date d4 = new Date();
        triTas(tab4);
        Date d5 = new Date();

        temps_fusion    += (d2.getTime() - d1.getTime());
        temps_insertion += (d3.getTime() - d2.getTime());
        temps_qs += (d4.getTime() - d3.getTime());
        temps_tas += (d5.getTime() - d4.getTime());
      }

    temps_fusion /= iteration;
    temps_insertion /= iteration;
    temps_qs /= iteration;
    temps_tas /= iteration;

    System.out.println(n + "\t" + temps_insertion + "\t" + temps_fusion
        + "\t" + temps_qs + "\t" + temps_tas);
```

```
                }
        }
}
```

Listings

4 Annexes - courbes expérimentales

Courbe 1 Résultats pour des tableaux de 0 à 200 éléments

Courbe 2 Résultats pour des tableaux de 0 à 700 éléments

Courbe 3 Résultats pour des tableaux de 0 à 6 000 éléments