Formation enseignement de l'Informatique– 2011

Algorithmes de Tri

Olivier Bournez

1 Tris par comparaisons

1.1 Tri par remontée de bulles

L'idée consiste à comparer deux éléments adjacents, et à les échanger s'ils ne sont pas dans l'ordre. Lorsqu'on réalise un passage, on range le plus grand élément en fin de tableau (ou le plus petit au début). On a donc réalisé une partition entre l'élément le plus grand d'une part, et tous les autres éléments d'autre part.

1. Dérouler l'algorithme sur le tableau suivant.

   $$[5, 8, 9, 4, 1]$$

   - $[5, 8, 9, 4, 1] -> [5, 8, 4, 9, 1] -> [5, 8, 4, 1, 9]$
   - $[5, 4, 8, 1, 9] -> [5, 4, 1, 8, 9] -> [4, 5, 1, 8, 9]$
   - $[4, 1, 5, 8, 9] -> [1, 4, 5, 8, 9]$

2. En supposant que le tableau à trier est $[2, 3, 4, \ldots, n, 1]$, combien fait-on d'échanges ? Combien fait-on de comparaisons ?

   Voici la liste des tableaux en fonctions des échanges
   - au départ : $[2, 3, 4, \ldots, n, 1]$,
   - ensuite : $[2, 3, 4, \ldots, 1, n]$,
   - $[2, 3, 4, \ldots, 1, n-1, n]$,
   - $[2, 3, 4, \ldots, 1, n-2, n-1, n]$,

   Au total, on fait $0(n^2)$ comparaisons et $n-1$ échanges

3. Ecrire en Java la méthode assurant cet algorithme.

```
 static void tribulle(int []t){
  AnimTri.showArray(a, MAX, "Selection");
    int n=t.length;
    for (int i=n-1; i>=0; --i)
for (int j=0; j<i; ++j)
        if (t[j+1] < t[j]) { // comparer deux voisins
    int tmp = t[j];
    t[j] = t[j+1];
    t[j+1] = tmp;
    AnimTri.showElement(j);
    AnimTri.showElement(j+1);
    // les echanger si necessaire
}
 }
```

1.2 Tri par sélection

Cela consiste à déterminer le plus petit élément, puis le deuxième petit élément, et ainsi de suite.

1. Dérouler l'algorithme sur le tableau suivant.

$$[5, 8, 9, 4, 1]$$

On note dans ce qui suit E_1 la partie déjà triée, et E_2 le reste.
- $E_1 = [1]$ et $E_2 = [8, 9, 4, 5]$
- $E_1 = [1, 4]$ et $E_2 = [9, 8, 5]$
- $E_1 = [1, 4, 5]$ et $E_2 = [8, 9]$
- $E_1 = [1, 4, 5, 8]$ et $E_2 = [9]$
- $E_1 = [1, 4, 5, 8, 9]$ et $E_2 = \emptyset$

2. Écrire en Java les méthodes assurant cet algorithme.

```
static void monTri() {
  AnimTri.showArray(a, MAX, "Selection");
  for (int i=0; i<N; i++){
    // on suppose que le ieme est le plus petit
    int min  = i;

    // sur tout ceux qui sont a droite de i on cherche si on trouve plus petit
    for (int j=i; j<N; j++){
      if (a[j]<a[min]){
        min = j;
      }
    }
    // a la sortie de la boucle le plus petit c'est min !

    // on permute i et min
    if (min!=i){
      int tmp = a[i];
      a[i] = a[min];
      a[min] = tmp;
      AnimTri.showElement(i);
      AnimTri.showElement(min);
    }
  }
}
```

3. Quelle est la complexité du tri par sélection ?

La sélection du minimum dans un ensemble à n éléments se fait en $O(n)$. L'opération est répétée $n-1$ fois, pour un ensemble dont la taille va de 2 à n. La complexité en temps du tri par sélection est donnée par $\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$.

1.3 Tri rapide

Inventé par HOARE en 1961, ce tri s'appelle aussi parfois tri par segmentation, ou quicksort. Le principe du tri est de créer une partition de la liste E de départ deux sous-listes E_1 et E_2, où E_1 sont les éléments plus petits qu'un élément, appelé "pivot", et E_2 ceux qui sont plus grands. La qualité de la partition dépend du choix du pivot. L'idéal serait de choisir comme pivot la médiane des n éléments de l'ensemble E de départ. Une solution simple et relativement efficace est de choisir la médiane comme le premier élément (ou celui du milieu, ou le dernier).

1. Dérouler l'algorithme sur le tableau suivant.

$$[8, 5, 9, 4, 1]$$

2. Écrire en java les méthodes nécessaires au tri rapide.

- Le premier pivot est 8
- il y a deux listes [9] et $[5, 4, 1]$
- Pour la liste $[5, 4, 1]$, le premier pivot est 4 et on obtient deux listes [1] et [5]

```
 static void quicksort(int []numbers, int low, int high) {
int i = low, j = high;
// On choisit un pivot
int pivot = numbers[low];

// Divide into two lists
while (i <= j) {
// If the current value from the left list is smaller then the pivot
// element then get the next element from the left list
while (numbers[i] < pivot) {
i++;
}
// If the current value from the right list is larger then the pivot
// element then get the next element from the right list
while (numbers[j] > pivot) {
j--;
}

// If we have found a values in the left list which is larger then
// the pivot element and if we have found a value in the right list
// which is smaller then the pivot element then we exchange the
// values.
// As we are done we can increase i and j
if (i <= j) {
    exchange(numbers,i, j);
i++;
j--;
}
}
// Recursion
if (low < j)
    quicksort(numbers,low, j);
if (i < high)
    quicksort(numbers,i, high);
}

    static void exchange(int []numbers, int i, int j) {
int temp = numbers[i];
numbers[i] = numbers[j];
numbers[j] = temp;
AnimTri.showElement(i);
        AnimTri.showElement(j);
}
```

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

  initialisation();
  impression();
   AnimTri.showArray(a, MAX, "Selection");
  quicksort(a,0,a.length-1);
  impression();
}
```

2 Utilisation des tableaux triés

2.1 Chercher un élément dans un tableau

Le but est d' écrire une fonction de recherche d'une valeur dans un tableau d'entiers.

1. Écrire une méthode `recherche(int[] t, int x)` qui prend un quelconque tableau t et une valeur entière x et qui retourne une position de x dans t. Si x n'est pas dans t, la méthode retourne -1.

   Donner le nombre de comparaisons dans le pire des cas.

   ```
   public int recherche(int[] t, int x){
       int temp =-1;
       for (int i = 0; i < t.length; i++) {
       if( t[i] ==x ){ tmp =i ;}}
   return tmp;}
   ```

   Le pire des cas est quand x n'est pas dans t. Il y a alors n comparaisons, où n est la taille du tableau t.

2. Dans le cas où le tableau est trié par ordre croissant, il est possible de retrouver plus rapidement la valeur recherchée x par dichotomie. Le principe est de comparer x à la valeur stockée au milieu du tableau :

   si elle est plus grande que x, il faut chercher dans la moitié inférieur du tableau, et sinon dans la moitié supérieure. et sinon dans la moitié supérieure.

   On recommence alors sur la portion du tableau de taille moitié et ainsi de suite jusqu'à trouver la valeur. Écrire en Java la méthode `rechercheDichotomie(int[] t, int x)` basée sur cette approche.

   Donner le nombre de comparaisons dans le pire des cas pour le cas où le nombre d'éléments du tableau est égal à 2^k.

   Voir prochain cours.

3. Comparer ces deux méthodes pour le cas où le nombre d'éléments du tableau est égal à 2^k.

– Première approche : $O(2^k)$ opérations.
– Deuxième approche : $O(k)$ opérations, mais il faut trier le tableau.

2.2 Comparaison des tableaux

On dit que deux tableaux sont
– égaux si ils contiennent les mêmes éléments aux mêmes positions
– similaires si ils contiennent les mêmes éléments mais pas forcément aux mêmes positions
– comparables si l'ensemble des valeurs qu'ils contiennent est le même.

Par exemple,
– les tableaux $[5, 2, 4, 1, 2, 1]$ et $[1, 1, 2, 2, 4, 5]$ sont similaires mais pas égaux.
– les tableaux $[5, 2, 4, 1, 2, 1]$ et $[1, 2, 4, 5]$ sont similaires mais pas égaux.

1. Écrire une méthode qui teste si deux tableaux sont égaux.

```
public int egalite(int[] t1, int[] t2){
int temp = 1;
if  (t1.length !=  t2.length ) return -1 ;

for (int i = 0; i < t.length; i++) {
    if( t1[i] !=t1[i] ){ tmp =0 ;}}

return tmp;}
```

2. Écrire une méthode qui teste si deux tableaux sont similaires. Donner la complexité de cet algorithme.

```
public int similaire(int[] t1, int[] t2){
if  (t1.length !=  t2.length ) return -1 ;
trier(t1);
trier(t2);
return egalite(t1,t2); }
```

   Complexité de cet algorithme = $O(nlogn)$ (à cause du tri).

3. Écrire une méthode qui teste si deux tableaux sont comparables.

```
public int[]  supprimer_doublon(int[] t)
// on suppose t non vide et trié
{
int[] t1 = new int [t.length]
int pointeur =1;
t1[0]=t[0];
for (int i = 1; i < t.length; i++) {
    if( t[i] !=t[i-1] ){ pointeur =pointeur +1; t1[pointeur]=t[i];  }}
int[] tt = new int [pointeur];
for (int i=0; i<pointeur;i++) {
tt[i]=t1[i];
}
return tt;
}

public int comparable(int[] t1, int[] t2){
trier(t1);
trier(t2);
supprimer_doublon(t1);
supprimer_doublon(t2);
return similaire(t1,t2); }
```

   Complexité de cet algorithme = $O(n\ \ logn)$

3 Accélération du tri par sélection : le tri arbre.

Imaginé par Williams en 1964, ce tri est aussi appelé tri en tas ou tri maximier. Ce tri est une accélération du tri par sélection et échange : une organisation préalable de l'ensemble de

départ permet d'obtenir une sélection du maximum en un temps qui est dans l'ordre de $log_2 n$, ce qui permet à l'algorithme de redevenir optimum. Le vecteur est considéré comme ayant une structure d'arbre binaire. On fera l'hypothèse que les éléments du tableau ont des indices qui vont de 1 à f. Un élément d'indice i du vecteur est considéré comme un nœud de l'arbre ; son fils gauche s'il existe se trouve à l'indice $2 * i + 1$ et son fils droit s'il existe se trouve à l'indice $2 * i + 2$. Ce nœud est une feuille si $2 * i + 1$ est strictement supérieur à f

1. Ecrire une fonction `descendre` permettant de réorganiser un tas dont seule la racine n'est pas à sa place. La racine se trouve à l'indice d.

```
public static void descendre( int []t , int d, int nMax){
   int fg, fd, fm;
   if(d*2+1<nMax) {
          fg = 2*d+1;
          fd = 2*d+2;
          if(fd>=nMax) {fm = fg; }
          else
             {if (t[fg] > t[fd] fm = fg; else fm = fd;}
        if( t[d] > t[fm]){return;}
             else{
                  int aux = t[d];
                  t[d] = t[fm];
                  t[fm] = aux;
                  descendre( t, fm, nMax);
                   }
     }
 }
```

2. Ecrire une fonction `remonter` permettant de remonter le dernier élément à sa place dans le tas formés par les éléments précédent.

```
public static void remonter( int []t , int n){
       if(n==1) return;
       else if( t[n-1] > t[n/2 - 1]){
              int aux = t[n-1];
              t[n-1] = t[n/2 - 1];
              t[n/2 - 1] = aux;
              remonter( t, n/2);
           }
}
```

3. L'organisation d'un tableau peut se faire de deux manières :
   – remonter tous les éléments, en partant du second, jusqu'au dernier.
   – descendre tous les éléments, en partant du milieu, vers le début.
   Ecrire ces deux fonctions.

```
public static void organiserTas( int []t , int n ){
 for(int i = 2; i<n; ++i) remonter(t, i);
}

public static void organiserTasBis( int []t , int n ){
       for(int i = n/2; i>=0; --i) descendre(t, i, n);
}
```

4. En utilisant les fonctions précédentes, proposer une procédure `triTas` assurant le tri par Tas.

```
public staticvoid trierTas( int []t , int n ){
      organiserTas( t, n);
      for( int i = n-1; i>0; --i ){
            echanger(t[0], t[i]);
            descendre(t, 0, i);
          }
}
```