TP 7: Tris. Corrigé

Informatique Fondamentale (IF121)

1er Décembre 2004

Exercice 1 (Tri par insertion)
Clairement le plus naturel, mais certainement pas le plus facile à écrire en java. C'est facile de repérer l'endroit `j` où il faut placer l'élément `tab[i]` (simple boucle while). C'est moins facile de l'y insérer, car il faut décaler tous les éléments déja trié à partir de `j` d'un cran vers la droite afin de laisser la place `j` libre.

Notez que l'on peut quand même réaliser ceci avec le code suivant, le point intéressant étant qu'à la fin de l'algorithme, c'est `tab` lui-même qui devient trié : je n'ai pas créé au départ un tableau-résultat qui au final serait la version trié du `tab` de départ. C'est ce qu'on appelle un tri *en place*, car il ne demande pas d'allouer de la mémoire supplémentaire pour renvoyer son résultat, tout se fait dans le tableau de départ. En revanche, comme vous le verrez un jour, ce n'est pas l'algorithme de tri le plus efficace, d'autres s'exécute avec une plus petite complexité (i.e. terminent plus vite).

Pour être plus précis, sa complexité (en moyenne comme dans le pire des cas) est en $O(n^2)$ (n étant `tab.length`), ce qui signifie qu'il existe une constante c telle que le temps d'exécution de l'algorithme est majoré par $c \cdot n^2$. On dit que la complexité est quadratique. Le pire cas est lorsque les éléments sont déjà triés. Notez qu'il existe un résultat théorique qui dit que n'importe quel algorithme de tri est au moins en $O(n \cdot \log n)$.

```
static double[] triInsertion(double[] tab){
    double aux;
    int positionminimum,i,j,k;
    for(i=0;i<tab.length;i++){
        j=0;
        aux=tab[i];
        while ((aux>tab[j])&&(j<i)) j++;
        for(k=i;k>j;k--) tab[k]=tab[k-1];
        tab[j]=aux;
    }
   return tab;
}
```

Exercice 2 (Tri par recherche de minimum, encore appelé *tri par sélection*)
C'est facile de repérer le minimum du tableau à trier (appelons-le `tab`), c'est facile de le mettre en première position du tableau trié. C'est moins facile de le retirer de `tab` pour y chercher l'élément qui vient après, le deuxième plus petit élément.

On peut penser à se souvenir de sa valeur et recommencer la recherche de minimum sur les éléments strictement plus grand que celle-ci. Cela ne marche pas quand il y a plusieurs fois le même élément dans `tab`.

On peut penser à créer un second tableau `t` de même taille que `tab`, rempli de booleéns, avec `t[i]` étant `true` si et seulement si `tab[i]` a déjà été entré dans le tableau trié, auquel cas il ne faut pas le prendre en compte pour la recherche du minimum suivant. C'est assez laid, et ça prend de la place en mémoire.

On peut penser à remplacer l'élément qu'on vient d'entrer dans le tableau trié par une valeur plus grande que toutes celles que contenait `tab` au début. Cela nécessite soit de connaitre ce genre d'informations sur `tab`, soit de commencer le programme par une recherche du maximum de `tab`.

Une solution beaucoup mieux que toutes celles-là :
certes il n'est pas facile de faire une recherche de maximum en sautant les éléments qu'on a déjà considérés, mais il est très facile de commencer la recherche de minimum à partir de la ième case de `tab` plutôt que de la première. Donc plutôt que de supprimer un élément que l'on vient de considérer, on va le *déplacer*. Par exemple, on va déplacer le premier minimum trouvé, appelons-le `tab[a]` vers la première case de `tab`, et recommencer la recherche de minimum à partir de la deuxième case.
Certes, il y avait quelqu'un dans la première case de `tab`, donc il faudrait décaler `tab[0]`,...,`tab[a-1]` d'un cran vers la droite et cela introduit une nouvelle boucle, donc de la complexité. Plus simple, on peut juste échanger le minimum `tab[a]` du tableau avec l'élément `tab[0]`. Ca marche au premier coup, mais à partir du 2ème, on risque ainsi de réintroduire vers la fin du tableau un élément `tab[0]` qui avait déjà été considéré. Donc après le premier coup il faut donc faire attention d'échanger le i^{eme} plus petit élément, disons `tab[ai]`, non pas avec `tab[0]` qui a déjà pu être traité, mais avec `tab[i]` qui n'a pas pu l'être. Ainsi, on recommence la recherche de minimum à partir de `tab[i+1]`.

Tout comme le tri par insertion, le code-ci après fait tout dans le tableau `tab`, c'est à nouveau une implémentation *en place*. Par contre, la complexité de l'algorithme n'est toujours pas très bonne. En moyenne comme dans le pire cas elle est aussi quadratique. Ici, le pire cas est lorsque les éléments sont triés dans l'ordre inverse.

```
static double[] triSelection(double[] tab){
    double min;
    int positionminimum,i,j;
    for(i=0;i<tab.length;i++){
        min = tab[i];
        positionminimum=i;
        for(j = i+1 ;j<tab.length;j++){
            if (tab[j] < min){
                min = tab[j];
                positionminimum = j;
                }
            }
        tab[positionminimum]=tab[i];
        tab[i]=min;
   }
   return tab;
}
```

Exercice 3 (Tri-bulle)
Clairement le plus facile à écrire en java. Notez que lui-aussi est un tri en place, mais sa complexité est parmi les plus mauvaises.

```
static double[] triBulle(double[] tab){
    int i=0;
    double aux;
    while(i<tab.length-1){
        if (tab[i]<=tab[i+1]) i++;
        else {
            aux=tab[i];
            tab[i]=tab[i+1];
            tab[i+1]=aux;
            i=0;
            }
        }
    return tab;
}
```

Exercice 4 (Quick-sort)
On commence les tris efficaces. Notez qu'en disant "répartissez les éléments de `tab` en deux tableaux plus petits, je n'ai pas précisé la taille de ces tableaux, ils peuvent être de longueur très inégales. Du coup, un moyen simple de répartir les éléments est de prendre un *pivot*, et de mettre dans `tab1` tous les éléments qui lui sont inférieur, et de mettre dans `tab2` tous ceux qui lui sont supérieur. Bien sûr, il faut que `tab1` et `tab2` aient chacun au moins un élément, donc un moyen pour en être sûr est de prendre comme pivot un des éléments de `tab` (et de le placer lui dans celui de des deux tableaux `tab1` et `tab2` qui est le plus petit).

Dans le code suivant, on prend `tab[0]` comme pivot. On commence par calculer les tailles respectives `n` et`m` de `tab1` et `tab2` en prévoyant pour le plus petit une case de plus pour y mettre le pivot. On crée les tableaux, puis on répartit les éléments en mettant le pivot dans la case qui a été prévue pour lui.

Bien entendu, dès que le tableau a moins de 1 élément, il est trié donc c'est lui qu'on renvoit. Sinon, on appelle récursivement la méthode QuickSort, sur les tableaux `tab1` et `tab2` dont on récupère les versions triées que l'on recolle ensemble par la méthode `concatene` du TP5.

```
static double[] QuickSort(double[] tab){
    if (tab.length<=1) return tab;
    else{
        int n=0,m=0;
        for (int k=1;k<tab.length;k++){
            if (tab[k]<tab[0]) n++;
            else m++;
            }
        boolean b=(n<m);
        if (b) n++;
        else m++;
```

```
        double[] tab1=new double[n];
        double[] tab2=new double[m];
        int i=0,j=0;
        for (k=1;k<tab.length;k++){
            if (tab[k]<tab[0]){
                tab1[i]=tab[k];
                i++;
                }
            else{
                tab2[j]=tab[k];
                j++;
                }
            }
        if (b) tab1[n-1]=tab[0];
        else tab2[m-1]=tab[0];
        return concatene(QuickSort(tab1),QuickSort(tab2));
        }
}
```

Notez qu'on aurait pu éviter de parcourir deux fois le tableau `tab` en disant que `tab1` et `tab2` avaient la taille `tab.length` mais n'étaient pas remplis jusqu'au bout. Cela qui ferait gagner un peu de complexité, mais utiliserait beaucoup plus de mémoire. Ensuite, c'est un choix personnel de savoir ce qui est le pire.

Notez que le QuickSort n'est pas un tri en place, il utilise en effet pas mal de mémoire. En revanche, le temps d'exécution moyen est en $O(n \cdot \log n)$. Le temps d'exécution dans le pire cas est pourtant quadratique. Le pire cas, c'est quand le tableau est déjà trié ou trié dans l'ordre inverse. Le meilleur cas, c'est quand à chaque étape le pivot est la médiane du tableau. On pourrait donc chercher la médiane de `tab` au lieu de prendre `tab[0]`. Malheureusement, cette recherche elle-même prend du temps, à peu près le même que celui qu'elle nous ferait gagner parce qu'elle trouve le bon pivot.

Exercice 5 (Tri-fusion)

Il y a peu de problèmes, tout le gros du travail a été fait dans la méthode `fusion` du TP5. Dès que le tableau a moins de 1 élément, il est trié donc c'est lui qu'on renvoit. Sinon, on appelle récursivement la méthode triFusion, sur les tableaux `tab1` et `tab2` dont on récupère les versions triées que l'on fusionne ensemble par la méthode `fusion` du TP5.

```
static double[] triFusion(double[] tab){
    if (tab.length<=1) return tab;
    else{
        double[] tab1=new double[tab.length/2];
        double[] tab2=new double[tab.length-tab1.length];
        int k;
        for (k=0;k<tab1.length;k++)tab1[k]=tab[k];
        for (k=0;k<tab2.length;k++)tab2[k]=tab[k+tab1.length];
        return fusion(triFusion(tab1),triFusion(tab2));
        }
}
```

Notez que le triFusion n'est pas un tri en place, il alloue de la mémoire. On peut économiser un peu de mémoire en ne créant pas les tableaux `tab1` et `tab2`, mais en indiquant où ils commencent et où ils finissent dans `tab`.

```
static double[] aux(double[] tab, int debut, int fin){
    if ((fin-debut)<=1){
        double[] res=new double[fin-debut];
        if (fin==(debut+1)) res[0]=tab[debut];
        return res;
    }
    else return fusion(aux(tab,debut,(debut+fin)/2),aux(tab,(debut+fin)/2,fin));
}
static double[] triFusion(double[] tab){
        return aux(tab,0,tab.length);
}
```

Même cette version n'est pas en place, car la méthode `fusion` alloue de la mémoire.

En revanche, le temps d'exécution, même dans le pire cas, est en $O(n \cdot \log n)$, c'est donc mieux que le QuickSort.

Maintenant, peut-on faire un tri qui soit à la fois en place et en $O(n \cdot \log n)$? ? ?