TD/TP n°5 – Tri de tableau

Structures de données (IF 122)

L'objet de ce TD est de manipuler différents algorithmes de tris et de comparer leur efficacité dans le *pires des cas*. On se donne à chaque fois un tableau d'entiers et le problème consiste à modifier l'ordre des éléments de telle sorte qu'ils soient triés après l'application de l'algorithme.

1 Tris par sélection

Le premier des algorithmes de tri est l'un des plus simples. Le principe est de sélectionner l'élément le plus petit du tableau, c'est-à-dire de trouver l'entier p tel que $\forall i, t[i] \geqslant t[p]$. Une fois cet emplacement trouvé, on échange les éléments $t[1]$ et $t[p]$. Puis on recommence ces opérations sur le reste du tableau (c'est à dire les éléments compris entre les indices 2 et n). On recherche alors le plus petit élément de cette nouvelle suite de nombre et on l'échange avec $t[2]$. Et ainsi de suite ... jusqu'au moment où l'on a placé tous les éléments du tableau.

Voici les états successifs du tableau $[9, 5, 1, 6, 2]$ par cette méthode :

9	5	1	6	2
1	5	9	6	2
1	2	9	6	5
1	2	5	6	9
1	2	5	6	9

Exercice 1 (TD) — *Complexité dans le pire des cas* On qualifie un tableau donné de *pire des cas* lorsqu'il fait effectuer à l'algorithme le nombre maximal d'opérations, le type d'opérations étant à préciser (allocation mémoire, calcul arithmétique, ...).

- Quel est le pire des cas en nombre d'échanges pour la méthode précédente ?
- À combien d'échanges procède-t-on alors ?
- Quel est le nombre de comparaisons effectuées lors du tri ?

Exercice 2 (TD/TP) — *Tri par sélection* Écrivez les méthodes suivantes :

1. une méthode `echange(int [] t, int p1, int p2)` qui échange les éléments `t[p1]` et `t[p2]` du tableau.
2. une méthode `triSel(int [] t)` itérative qui réalise le tri par sélection.

2 Tri à bulle

Le **tri à bulles** est sans doute le plus lent des tris possibles. L'idée est de comparer successivement tous les éléments adjacents d'un tableau et de les échanger si le premier élément est supérieur au second. On recommence cette opération tant que tous les éléments ne sont pas triés

Exercice 3 (TD) — *Complexité* On s'intéresse à la complexité en terme d'échanges

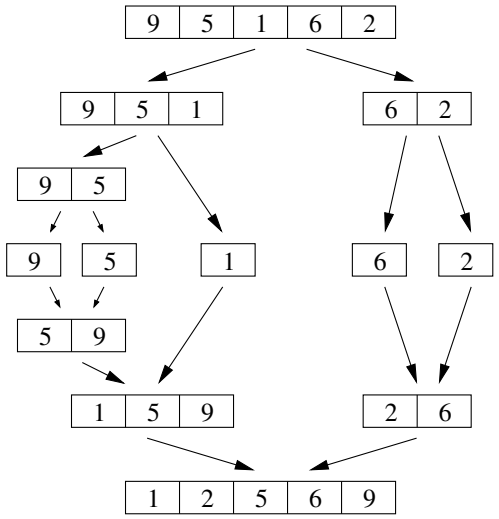
- Testez cette méthode à la main sur le tableau de l'exercice précédent.
- Quel est le pire cas possible et quelle est alors la complexité de l'algorithme.

Exercice 4 (TP) — *Tri à bulles* Implantez une méthode `triBulle(int [] tab)` qui trie le tableau `tab` sur place (c'est à dire sans tableau auxiliaire) par la méthode du tri à bulle.

3 Tri fusion

Le **tri par fusion** est le dernier exemple d'algorithme de tri que nous verrons dans ce TD. Le principe repose sur la constatation qu'il est facile de fusionner deux tableaux déjà triés. L'algorithme consiste donc à diviser le tableau en deux parties à peu près égales, à trier les deux moitiées et à fusionner le résultat.

Par exemple, si le tableau à trier est $[9, 5, 1, 6, 2]$, l'algorithme donne la succession suivante :



Exercice 5 (TD/TP) — ***Division*** Donnez une méthode `divise(int [] tab, int pos1, int pos2)` retournant un tableau de longueur `pos2` − `pos1` +1 contenant les valeurs de `tab` entre les valeur `pos1` et `pos2` inclus.

Exercice 6 (TD/TP) — ***Fusion*** Donnez une méthode `fusionne(int [] a,int [] b)` retournant la fusion de deux tableaux triés `a` et `b`.

Exercice 7 (TD/TP) — ***Tri fusion*** À partir des deux méthodes précédentes, proposez une méthode `triFusion(int [] tab)` **récursive** retournant le tableau `tab` trié dans l'ordre croissant.

4 Autour des algorithmes précédents

Exercice 8 (TD) — ***Complexité du tri fusion*** Donnez la complexité de l'algorithme du tri fusion en terme de comparaisons dans le pire des cas.

Exercice 9 (TP) — ***Amélioration du tri par sélection*** Adaptez le tri par sélection de manière à placer à chaque passage le minimum des valeurs non triées du tableau en début et le maximum de ces valeurs à la fin. Donnez en une version récursive.

Exercice 10 (TP) — ***Tri fusion sur place*** L'algorithme de tri fusion tel qu'il est présenté dans la section 3 possède l'inconvénient de créer un nombre non négligeable de copies des valeurs de départ (cf. la méthode `divise`). On peut se passer de ces copies en considérant que scinder le tableau en deux moitiées revient à donner les bornes correspondant aux cases que l'on veut trier.

Proposez une méthode `triFusionAux(int [] tab,int debut, int fin)` **récursive** retournant un tableau trié de longueur `fin` − `debut` +1 contenant les valeur de `tab` entre les cases `debut` et `fin` (inclues). Modifiez `triFusion` en conséquence.

▶ **Exercice 1**

```
public static int [] fusionne(int [] tab1, int [] tab2){
 int i = 0;
 int j = 0;
 int k = 0;
 int [] tab = new int[tab1.length+tab2.length];
 // On commence par effectuer la fussion
     while ((i<tab1.length) && (j<tab2.length)){
         if (tab1[i]<tab2[j]) {
             tab[k]=tab1[i];
             i++;
         }
         else
         {
             tab[k]=tab2[j];
             j++;
         }
         k++;
     }

     // Les deux whiles suivants sont la pour finir le tableau dont
     // on a pas encore atteind la borne
     // L'un des deux n'est pas executer
     while (i<tab1.length) {
         tab[k]=tab1[i];
         i++;
         k++;
     }
     while (j<tab2.length) {
         tab[k]=tab2[j];
         j++;
         k++;
     }
 return tab;
 }
```

▶ **Exercice 2**

Cette version est volontaire tres tres mauvaise en complexité (à cause des recopiage de tableau). Elle presente cependant l'avantage d'être a mon avis plus "simple intellectuellement" puisque l'on a pas besoin de s'occuper des bornes du tableau.

On peut faire beaucoup mieux en ne passant pas notre vie a recopier le tableau en argument mais en donnant les bornes du "vrai " tableau (cf exo suivant) Pensez tous a me traiter de mauvais aupres de vos etudiants.

```
public static int [] triFusion(int [] tab){
    if (tab.length > 1){
        // On commence par determiner le "milieu" de tab
        int milieu = tab.length /2;
        // Pour eviter de se trimbaler les indice on recopie
        // les deux tableaux
        int [] tab1 = new int[milieu];
        for (int i =  0;i<milieu; i++){tab1[i]=tab[i];}
        // Attention aux taille impaires
        int [] tab2 = new int[tab.length - milieu];
        for(int i = milieu;i<tab.length;i++){tab2[i-milieu]=tab[i];}
```

```
        // Puis on tri les deux tableaux
        tab1 = triFusion(tab1);
        tab2 = triFusion(tab2);
        // Enfin on fusionne
        return(fusionne(tab1,tab2));
    }
    else {return tab;}
}
```

▶ **Exercice 3**

```
public static int [] triFusionAux(int [] tab, int debut,int longueur){
    if (longueur > 1){
        // On commence par determiner le "milieu" de tab
        int milieu = debut + longueur /2;
        int [] tab1 = triFusionAux(tab,debut,milieu-debut);
        int [] tab2 = triFusionAux(tab,milieu,longueur-longueur/2);
        // Enfin on fusionne
        return(fusionne(tab1,tab2));
    }
    else {
        int [] tab2 = {tab[debut]};
        return tab2;
    }
}


public static int [] triFusion(int [] tab){
    return(triFusionAux(tab,0,tab.length));
}
```

▶ **Exercice 4**

```
public static void echange (int [] tab, int position1, int position2) {
    int temp = tab[position1];
    tab[position1] = tab[position2];
    tab[position2] = temp;
}
```

▶ **Exercice 5**

```
public static int partitionne1 (int [] tab){
    int valeur = tab[0];
    // On commence par mettre le pivot au bout
    echange(tab,tab.length-1,0);
    int place_pivot = 0;
    // On calcule alors la place finale du pivot
    // Tout en faisant les echanges necessaires
    for (int i = 0; i < tab.length -1;i++){
        if (tab[i]<=valeur) {
            echange(tab, place_pivot,i);
            place_pivot++;
        }
    };
    // On met le pivot en place
```

```
    echange(tab,place_pivot,tab.length-1);
    // Et on retourne la place du pivot
    return place_pivot;
}
```

▶ **Exercice 6**

```
public static int partitionne (int [] tab, int debut, int longueur){
    int valeur = tab[debut];
    // On commence par mettre le pivot au bout
    echange(tab,debut+longueur-1,debut);
    int place_pivot = debut;
    // On calcule alors la place finale du pivot
    // Tout en faisant les echanges necessaires
    for (int i = 0; i < longueur -1;i++){
        if (tab[debut+i]<=valeur) {
            echange(tab, place_pivot,debut+i);
            place_pivot++;
        }
    };
    // On met le pivot en place
    echange(tab,place_pivot,debut+longueur-1);
    // Et on retourne la place du pivot
    return place_pivot;
}
```

▶ **Exercice 7**

```
public static void triRapideAux(int [] tab,int debut, int longueur){
    int fin = debut + longueur;
    if ((longueur <= 1) || ( fin > tab.length) || (debut <0)) {return; }
   else{
        int emplacement_pivot = partitionne(tab,debut,longueur);
        triRapideAux(tab,debut,emplacement_pivot-debut);
        triRapideAux(tab,emplacement_pivot+1,longueur + debut  - emplacement_pivot -1);
    }
}

public static void triRapide(int [] tab){
    triRapideAux(tab, 0 , tab.length);

}
```

▶ **Exercice 8**

```
public static void triBulle(int [] tab){
    int tmp;
    int n = tab.length;
    for (int i=0; i<n-1; i++) {
        for (int j=0; j<n-1-i; j++)
            if (tab[j+1] < tab[j]) {
                tmp = tab[j];
                tab[j] = tab[j+1];
                tab[j+1] = tmp;
            }
    }
}
```