Cours 10 du 3 avril :

Tris récursifs

Tri-fusion (merge-sort)

(cette partie a été traitée le 27 mars)
Il s'agit d'une application de "diviser pour régner"
Principe:

pour trier un tableau `tab[l,..,r]` :
    si `l=r`, le tableau est trié
    sinon
        choisir m dans `[l,r]`
        trier `tab[l,..,m]`
        trier `tab[m+1,r]`
        fusionner dans `tab[l,r]` `tab[l,..,m]` et `tab[m+1,r]`

L'opération fusionner consiste à construire un tableau trié à partir de deux (sous)-tableaux triés.

Fusion:

Partant de
    tableau `a:` `a[al…ar]` tel que `a` est ordonné
    tableau `b:` `b[bl…br]` tel que `b` est ordonné
Construire
    tableau `res:` `res[l..r] (r=l+al-ar+bl-br)`
tel que:
    `res` est ordonné
    `res` est la fusion de `a` et de `b`

(`res` est la fusion de `a` et `b` signifie informellement que chaque élément de `res` provient d'exactement un élément de `a` ou de `b`, formellement c'est un peu lourd. Par exemple:
    il existe `f:` injective `[al,ar]-> [l,r]` tel que pour tout `x` appartenant `[al,ar]`
`res[f(x)] = a[x]`
    et il existe g injective : `[bl,br]-> [l,r]` tel que pour tout `x` appartenant `[bl,br]`
`res[g(x)] = b[x]`
    telles que l'intersection de `f(([ar,al]` et de `g([b,br])` est vide)

A partir de l'invariant:
    (I) `res[l,k[` *est la fusion ordonnée de* `a[al,i[` *et de* `b[bl,j[`

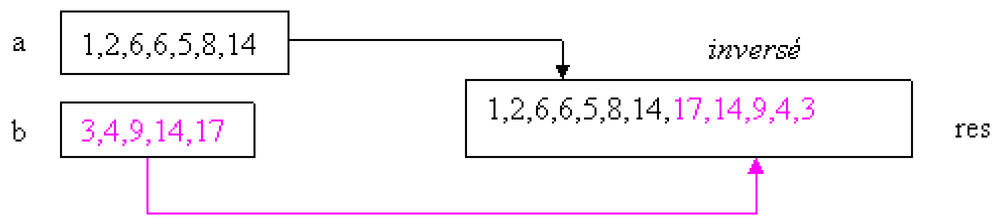
Avec une boucle sur `k`, on peut maintenir l'invariant en ajoutant à `res` le minimum entre *`a[i]`* et `b[j]` et en incrémentant l'indice `i` ou `j` correspondant. On obtient (en évitant les dépassements d'indices):

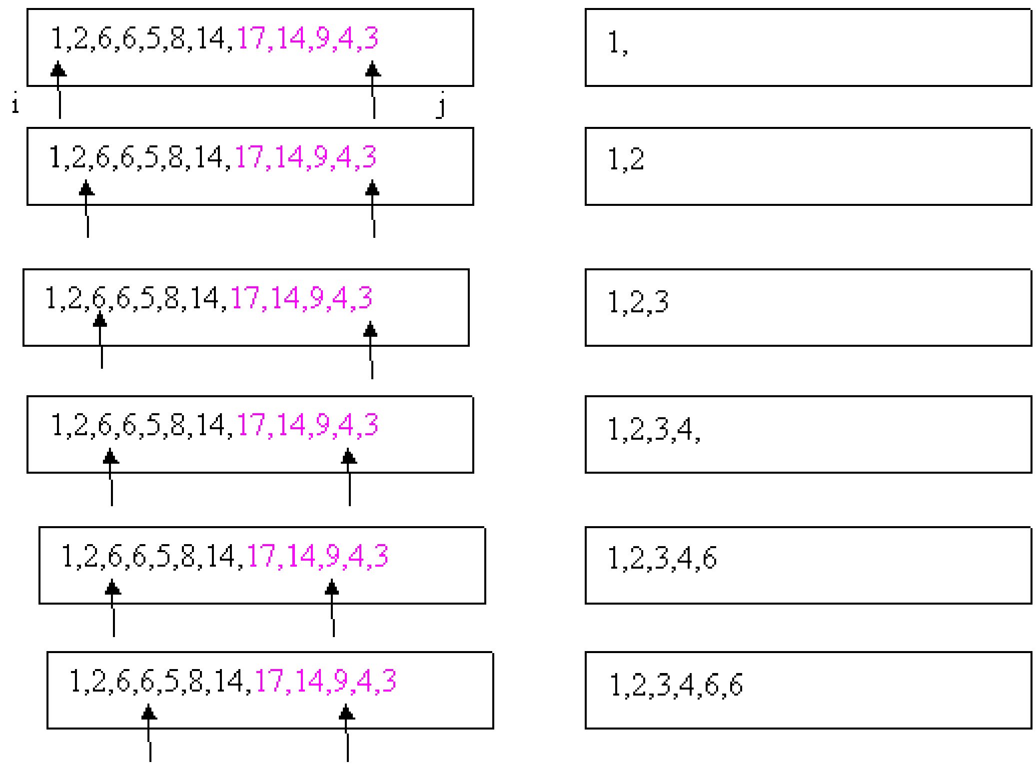
```
// fusion des tableaux a[al,ar] et b[bl,br]
// dans res[l, ...]
static void merge(int[] res, int l,
        int [] a, int al, int ar,
        int[] b,int bl,int br){
    int i=al;
    int j=bl;
    for(int k=l; k<=l+ar-al+br-bl+1;k++){
        if(i>ar){res[k]=b[j++]; continue;}
        if(j>br){res[k]=a[i++]; continue;}
        if(a[i]<b[j])res[k]=a[i++];
        else res[k]=b[j++];
    }
}
```

On fait une comparaison entre éléments du tableau à chaque itération et donc:

le nombre de comparaisons entre éléments du tableau est N si N est la taille du tableau résultant *`(ar-al+br-bl)`*

On peut aussi utiliser une technique plus astucieuse qui évite de vérifier les dépassements d'indices: Partant de a et de b trié, on copie dans aux a suivi de b *à l'envers,* on fait ensuite la fusion ordonnée avec deux indices l'un sur le début et l'autre sur la fin:





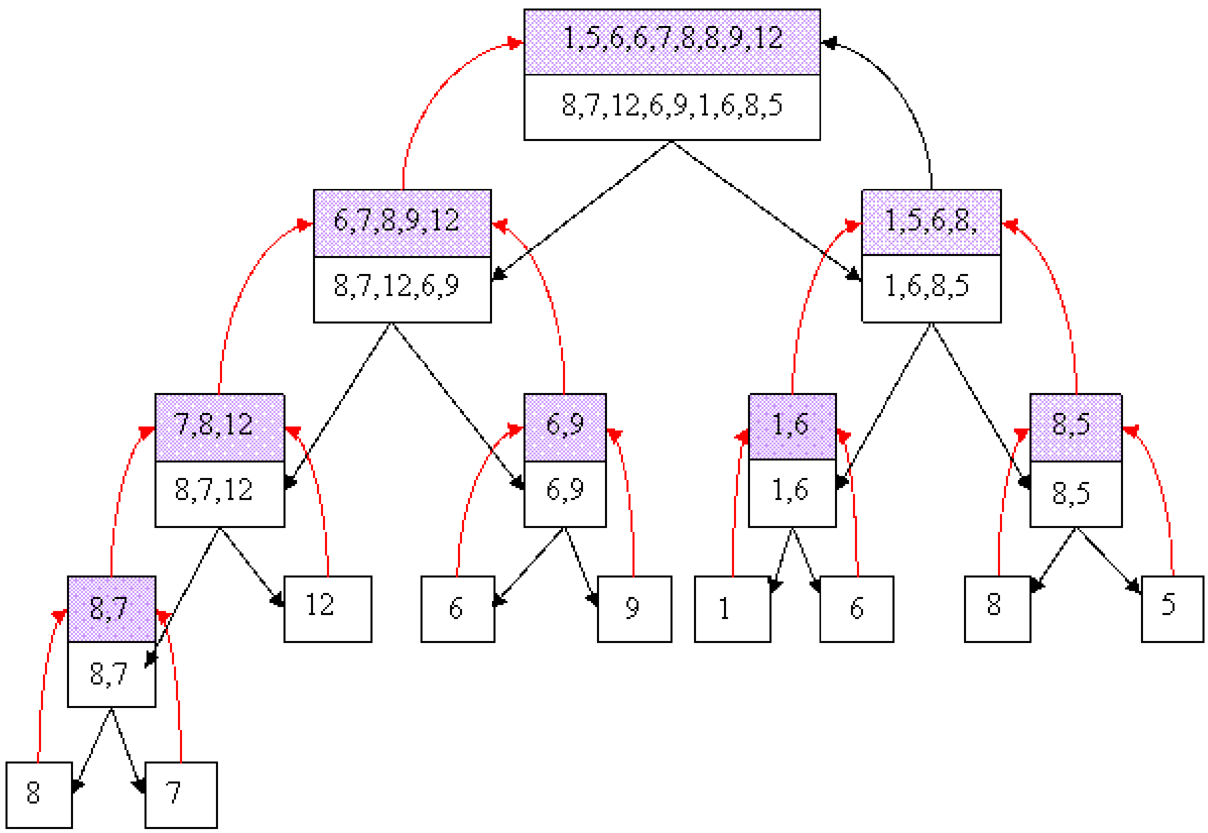
En travaillant avec les mêmes tableaux pour `a`, `b` et `t` (a est `t[l,m]` et `b t[m+1,r]` et le résultat est mis dans `t[l,r])` on obtient:

```
    // version de la fusion avec un seul tableau et ordre bitonic
    // copie du premier tableau (t[l,m]) dans aux
    // copie inversée du deuxième tableau (t[m+1,r]) à la fin de
aux
    // fusion dans t par compraison des éléments aux deux
extémités de aux
    // utilise un tableau auxiliaire aux
    public static void fusion(int[]t,int l,int m, int r){
        int i,j;
```

```
        for(i=m+1; i>l;i--) aux[i-1]=t[i-1];
        for(j=m;j<r;j++)aux[r+m-j]=t[j+1];
        for(int k=l; k<=r;k++)
            if(aux[j]<aux[i])t[k]=aux[j--];
            else t[k]=aux[i++];
    }
```

Tri fusion:

.



En utilisant brutalement la première méthode (merge) on obtient:

```
    // tri-fusion: version de base
    // principe: si le tableau a un seul élément il est trié
    //           sinon trier les deux moitiés du tableau et
fusionner
    // utilise un tableau auxiliaire
    public static void mergesort(int[] t,int l , int r){
        if(r<=l)return;
```

```
        int m=(r+l)/2;
        mergesort(t,l,m);
        mergesort(t,m+1,r);
        int []aux=new int[r-l+1];
        merge(aux,l,t,l,m,t,m+1,r);
        for(int i=l;i<=r;i++)
            t[i]=aux[i];
    }
```

Cependant on peut remarquer:

- chaque appel travaille sur des sous tableaux distincts de la méthode appelante (diviser pour régner!)
- on peut donc partager les mêmes tableaux
- `int []aux=new int[r-l+1];` n'est pas nécessaire: il suffit de créer une seule fois objet tableau de la taille totale pour aux dans une méthode appelée avant le tri.
- la copie de aux n'est pas nécessaire: seule la fusion nécessite d'utiliser un tableau distinct.

A partir de ces remarques on obtient:

```
    // a est le résultat du tri du tableau b
    // b sert aussi de tableau intermédiaire
    public static void mergesortV2(int [] a, int [] b, int l,int 
r){
        if(r<=l)return;
        int m=(r+l)/2;
        mergesortV2(b,a,l,m);
        mergesortV2(b,a,m+1,r);
        merge(a,l,b,l,m,b,m+1,r);
    }

    public static void mergesort(int [] t){
        aux=new int[t.length];
        for(int i=0;i<t.length;i++)aux[i]=t[i];
        mergesortV2(t,aux,0,t.length-1);
    }
```

En utilisant la forme améliorée de la fusion (méthode `fusion`) on aura:

```
    public static void mergesortV1(int[] t,int l , int r){
        if(r<=l)return;
        int m=(r+l)/2;
        mergesortV1(t,l,m);
```

```
        mergesortV1(t,m+1,r);
        fusion(t,l,m,r);
    }
    static void mergesortb(int t[]){
        aux=new int[t.length];
        mergesort(t,0,t.length-1);
    }
```

Évaluation: (partie du cours traitée le 3 avril)

le tri fusion de N éléments fait de l'ordre de N Log(N) comparaisons

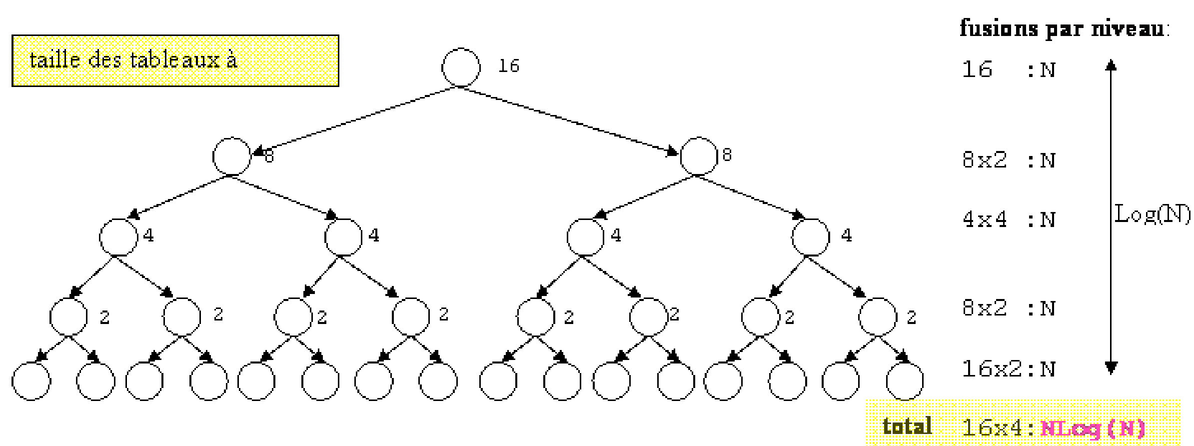
Comptons le nombre de comparaisons: la fusion d'un tableau de taille i et d'un tableau de taille j fait i+j comparaisons. Si M(N) est le nombre de comparaisons pour le tri-fusion de N éléments, on a (en supposant N pair, dans le cas impair N/2 est remplacé par la partie entière basse et la partie entière haute de N/2)

$$M(N)=M(N/2)+M(N/2)+N$$
$$M(1)=0$$

(Les deux premiers termes correspondent aux appels de tri-fusion sur la première et seconde entrée du tableau, le troisième terme correspond à la fusion de ces deux tableaux).

Pour $N=2^K$ la solution est N Log(N) (par récurrence). Dans le cas général on peut montrer que c'est de l'ordre de N Log(N)

On peut aussi retrouver ce résultat en regardant l'arbre des appels:



Remarque :

- l'ordre initial du tableau ne modifie pas les appels: le pire cas et le meilleur cas provoquent autant de comparaisons
- le cas moyen est égal au pire cas et u meilleur cas : N Log(N)
- on n'a utilisé qu'un tableau de auxiliaire de taille N

Ce tri se fait en N Log(N) comparaisons au lieu des N^2 pour le tri par insertion et par sélection.

Si on veut chercher M éléments dans une table de taille N, on peut:

1. utiliser la recherche linéaire: pour chercher M éléments on aura dans le pire cas on le fera en NM
2. trier le tableau et faire ensuite des recherches dichotomiques:

   dans le pire cas NLog(N) pour le tri + MLog(N) pour les recherches: (N+M)Log(N)

Dès que M est grand, (par exemple si M est de l'ordre de M) la deuxième méthode est plus efficace.

Tri-fusion version itérative

En considérant l'arbre des appels pour le tri-fusion on peut aussi déduire une solution itérative:
Supposons que l'arbre soit de hauteur k, définissons le niveau i comme étant les noeuds dont la hauteur est k-i (les feuilles sont de niveau 0 et la racine est de niveau k),

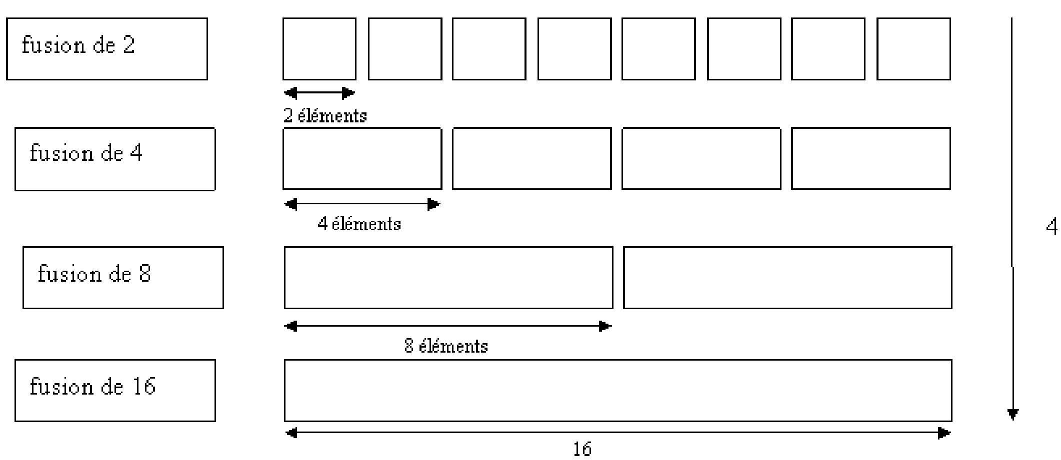
faire la fusion des N/2 groupes de 2 feuilles dans le tableau (on traite les noeuds de niveau 1) on obtient N/2 tableaux

…

faire la fusion 2 par 2 des 2^k tableaux du niveau précédent on obtient 2^{k-1} tableaux

…

jusqu'à ce qu'il n'y ait plus qu'un seul tableau.




```
// tri-fusion version itérative
```

```
    // "bottom-up"
    // utilise un tableau auxiliaire pour fusion
    public static void mergesortIter(int[]t){
        int r=t.length;
        for(int m=1;m<r;m=2*m)
            // m=1,2,...,2**k,.. de k=0 à k=log(r)
            for(int i=0; i<r-m;i+=2*m)
                // pour m=2**k regroupement en paquets de
2**(k+1) (=2m)
                // les t[i,i+2*m] sont les parties du tableau à
trier
                // correspondant au niveau log(m) dans l arbre
des appels
                // le milieu entre i et i+2*m est i+m-1
                // (attention à ne pas dépasser les bornes du
tableau)
                fusion(t,i,i+m-1,(i+2*m<r-1)?(i+2*m-1):r-1);
    }
}
```

Tri-rapide (quicksort)

Principe pour trier `t[l,r]`

- si `l=r` le tableau est trié
- sinon

    choisir un élément du tableau : `t[r]`. Soit `v=t[r]`
    partitionner `t[l,r-1]` en deux sous tableaux `t[l,i-1]` `t[i+1,r]` tel que tous les éléments de `t[l,i]` sont <= à `v` et tous les éléments de `t[i+1,r]` sont `>=v`
    mettre `v` dans dans `t[i]`
    appliquer récursivement le tri aux deux sous tableaux.

A chaque itération l'élément choisi est mis à sa position finale (tous ceux qui sont plus inférieurs ou égaux sont avant et ceux qui sont supérieurs ou égaux sont après)

```
    public static void quicksort(int []t, int l,int r){
        if (r<=l)return;
        int p=partition(t,l,r);
        quicksort(t,l,p-1);
        quicksort(t,p+1,r);
```

```
}
public static void echange(int[]t,int i,int j){
    int tmp=t[i];
    t[i]=t[j];
    t[j]=tmp;
}
// partition de t[l,r]
// prendre t[r] comme pivot
// mettre tous les éléments <= t[r] dans t[l,x]
// mettre tous les éléments >= à t[r] dans t[x+1,r]
// retourner x
static int partition(int []t,int l, int r){
    int i=l-1;
    int j=r;
    int p=t[r];
    while(true){
        while (t[++i]<p);
        // i indice du premier à droite >= p
        while (p < t[--j])if (j==l) break;
        // j indice du premier à gauche <= p
        if (i>=j)break;
        // échanger ces éléments
        echange(t,i,j);
        // tous les t[l,i] sont inférieurs ou égaux à p
        // tous les t[j,r] sont supérieurs ou égaux à p
    }
    // tous les t[l,i[ sont inférieurs à p
    // tous les t]j,r-1] sont supérieurs à p
    // (i==j) ou (j==l)
    echange(t,i,r);
    // tous les t[1,i] sont inférieurs ou égaux à p
    // tous les t[i+1,j] sont supérieurs ou égaux à p
    return i;
}
```

Evaluation:

Dans le pire cas le pivot peut être le plus grand élément du tableau et donc la partition est triviale. Le pire cas correspond donc à un tableau initialement trié. Le nombre de comparaisons est alors de N + (N-1) + …+1=(N+1)N/2.

Le meilleur cas correspond au cas où à chaque fois le pivot partitionne en deux tableaux de taille la moitié, dans ce cas on a la récurrence T(n)=2T(n/2)+N. Dont la solution est de l'ordre NLog(N)

Le cas moyen est plus complexe et donne aussi N Log(N) comparaisons en moyenne

Structures récursives: listes chaînées

A propos des tableaux:

- L'insertion dans un tableau d'un élément peur entraîner un déplacement de tous les éléments du tableau (N)
- L'accès aux éléments se fait en temps constant.
- Un tableau est une structure statique avec une taille fixe
-

Les listes chaînées permettent d'éviter ces problèmes, on aura:

- insertion et suppression en temps constant
- Mais l'accès à un élément se fait en temps proportionnel à la talle de la liste

Une liste constituée de Noeuds

```
class Noeud {
    Object item;
    Noeud suivant;
    public Noeud(Object o) {
        item=o;
        suivant=null;
    }
    public Noeud(Object o, Noeud n){
        item=o;
        suivant=n;
    }
}
```

- `Object`: tout objet peut être considéré comme étant de la classe Object, on récupère l'objet avec son type par un forçage de type:
  ```
  Type x=new Type();
  Object ob=x;
  Type y=(Type)x;
  ```
- Pour les types primitifs `(int`, `char` etc…) il existe des classes correspondantes qui permettent de passer d'un type primitif à un type référence, par exemple à `int` correspond la classe `Integer` :

  ```
  int k=100;
  Integer j,i=new Integer(100);
  Object ob=i;
  j=i;
  k=k+i.intValue();
  k=k+j;
  ```