TD4: tris

Tris élémentaires

Vous avez déjà vu un tri élémentaire: le tri par sélection dont la complexité dans le meilleur et le pire des cas est quadratique (i.e. en $\theta(n^2)$). Le tri à bulles est un autre tri quadratique, son nom s'inspire des bulles du champagne qui remontent en haut du verre. Ici, le plus grand élément remonte à droite du tableau par comparaison d'éléments successifs 2 à 2.

1. **Tri à bulles.** Complétez et ajoutez la méthode "triBulle" à la classe "TableauGenerique" en respectant les propriétés P1 et P2. Ce tri est appelé "tri à bulle" car comme les bulles de champagne en haut du verre, l'élément le plus grand remonte sa place à la fin de la boucle for.

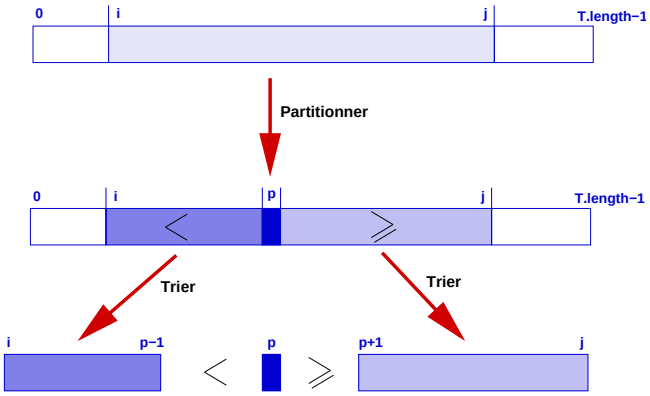
   Calculer sa complexité dans le meilleur et le pire des cas et donner des exemples de tableaux pour lesquels ces complexités sont atteintes.

```
public void triBulle() {
    boolean fini = false;
    int j = this.leTableau.length -1;
    while (!fini) {
        fini=true;
        for (int i=0; i<j; i++) {
            if (this.leTableau[i].compareTo(this.leTableau[i+1])>0) {
                ......................
                ......................
                ......................
                fini = false;
            }
        }
        // P1 : tab est trié de j à leTableau.length -1
        // P2 : les éléments de 0 à j-1 sont inférieurs aux éléments de j à tab.length-1
        j--;
    }
 }
```

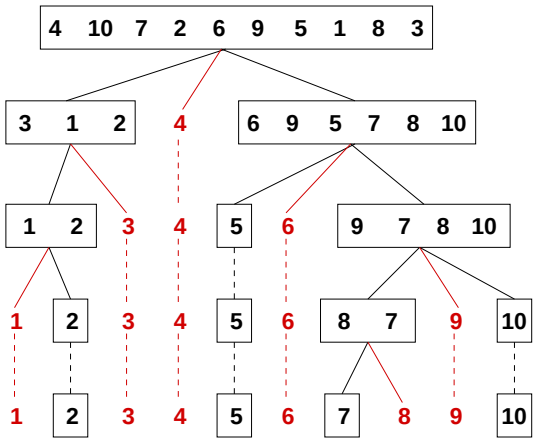
Tri rapide

Le tri rapide, ou *quicksort* est un tri très fameux dont la complexité en moyenne est en $nlog(n)$ (mais qui peut dégénérer en n^2). C'est la meilleure complexité que l'on puisse atteindre pour un tri. Ce tri combine une étape de partition qui coupe le tableau en 2 parties telles que tous les éléments de gauche sont inférieurs à ceux de droite, et un appel récursif pour continuer à traiter les 2 sous-parties.

Le schéma général du quicksort est ci-dessous (© cours algorithmique SI3 de Marc Gaëtano).



La figure suivante illustre le fonctionnement de ce tri sur un exemple, quand on partitionne en prenant le 1er élément e comme pivot c'est à dire de façon à ce qu'on ait à gauche les éléments plus petits que e et à droite ceux qui sont plus grands.



Comme ce tri est assez complexe, nous commençons par quelques exercices d'échauffement ...

2. **Salade de fruits.** On vous demande d'écrire une petite hiérarchie de classes (héritage) où l'on a des fruits, des fruits qui sont des agrumes (Ex : citron, pamplemousse, orange, clémentine) et des fruits à noyaux (Ex : prune, pêche, cerise, abricot). On a un panier de fruits dans lequel on peut créer de façon aléatoire plusieurs agrumes et plusieurs fruits à noyaux. Les fruits de ce panier peuvent être ordonnés en mettant à gauche les agrumes et à droite les fruits à noyaux.

   Attention: pour ordonner le panier, il ne faut pas utiliser un autre panier, il faut échanger les fruits 2 à 2 et chaque fruit doit bouger au maximum une fois.

   On vous donne la classe "Panier" à compléter et une classe de test. Vous pouvez tester votre code sur des exemples dans moodle.

3. **Partition.** Il s'agit maintenant d'écrire la méthode qui partitionne en 2 un tableau en prenant le 1er élément comme pivot. Cette méthode est une boucle similaire à celle pour ordonner le

panier de fruits. Mais comme cette méthode sera appelée successivement sur des sous-parties du tableau, cette méthode travaille entre les indices i et j. Pour bien réfléchir, il est conseillé de faire en sorte qu'à chaque étape la propriété suivante soit respectée :

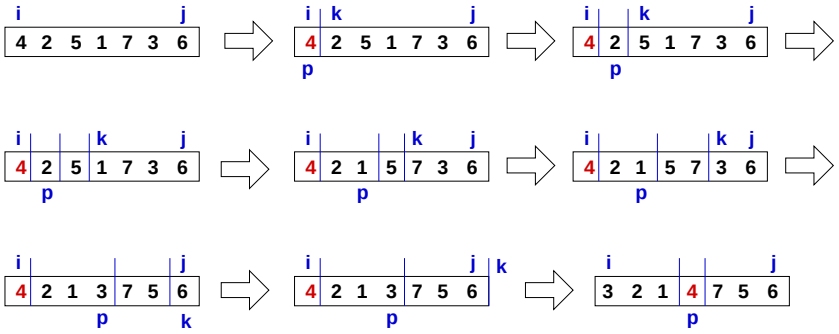
$$\forall g, i \leq g < p, \forall d, p < d \leq j, T[g] < T[p] \leq T[d]$$

avec

$$i \leq p \leq j$$

Cela est schématisé ci-dessous. L'élément pivot est $T[i]$, il est en rouge. La partie rose clair à gauche contient les éléments strictement inférieurs à $T[i]$, la partie rose foncé au milieu contient les éléments supérieurs à $T[i]$. La partie avec le ? est la partie du tableau qui n'a pas encore été traitée. La question est donc de mettre l'élément $T[k]$ à la bonne place, en l'échangeant avec un autre élément déjà traité.

Le fonctionnement de la partition est illustré ci-dessous sur un exemple.



Compléter la méthode private int partitionner(int i, int j) de la classe *QuickSort*. Elle partitionne le tableau entre ses indices de i à j et renvoie l'indice de séparation entre les éléments plus petit que le pivot et plus grands que le pivot, le pivot étant à sa place.

4. **Le quicksort**. Il ne vous reste plus maintenant qu'à compléter la méthode *trier* qui à chaque étape appelle partition pour partitionner le tableau puis s'applique récursivement sur les 2 parties obtenues (l'élément pivot reste à sa place).

5. **Complexité du quicksort**. Le nombre d'appels récursifs du quicksort dépend de la position du pivot à chaque étape de partition. Que se passe-t-il si le tableau est déjà trié en ordre croissant ? Que se passe-t-il si le tableau est déjà trié en ordre décroissant ? Que se passe-t-il si les éléments du tableau sont tous égaux ? Quel serait le cas idéal ? A chaque fois que c'est nécessaire, donner une relation de récurrence permettant de calculer la complexité.

6. **Application.** Quelqu'un a renversé le dictionnaire anglais qui n'est plus ordonné! Récupérez le dictionnaire "wordsAlpha-unsorted.txt". Appliquez le quicksort pour ré-ordonner ce dictionnaire, appliquez le tri par sélection, comparez les performances. Nota: il vous faut lire le dictionnaire et le transformer en tableau générique comme pour le correcteur.