Université de Lille
IUT de Lille

TP 14

R1.01 - Initiation au développement
BUT Informatique
Yann.Secq@univ-lille.fr
Fabien.Delecroix@univ-lille.fr
2022–2023

Objectifs: Savoir écrire des fonctions récursives simples.

Exercice 1 : Fonction récursive pour compter le nombre d'occurrences d'une lettre

Nous souhaitons dans cet exercice décompter le nombre d'occurrences d'une lettre dans une chaîne de caractères. Bien évidemment, nous voulons écrire un sous-programme effectuant ce calcul de manière récursive.

1. Définissez une fonction récursive directe (cf. votre cours) calculant le nombre d'occurrences d'une lettre donnée au sein d'une chaîne de caractères.
2. Définissez une fonction récursive terminale équivalente. Pour cela vous devez introduire une fonction auxiliaire qui aura un paramètre supplémentaire servant à retenir le nombre d'occurrences déjà rencontrées.

Exercice 2 : Recherche par dichotomie

Vous continuez ici à vous exercer à la récursivité avec un problème que nous avons déjà résolu à l'aide d'une fonction utilisant une boucle. Il s'agit de la recherche par dichotomie d'une valeur donnée dans un tableau trié donné.

Rappelons que la fonction `int rechDico (int[] tab, int val)` doit retourner l'indice de la valeur `val` dans tab, ou -1 si `val` n'apparait pas dans `tab`. On considère les valeurs du tableau comme n'apparaissant qu'une seule fois. Cette fonction utilise le principe *diviser pour régner*, ce qui lui permet d'être très efficace. C'est aussi la raison pour laquelle elle peut s'écrire facilement en utilisant la récursivité.

Rappelons l'algorithme de recherche dichotomique, en donnant ses équations de récurrence de manière semi-formelle.

$$rechDicho(tab, val) = \begin{cases} -1 & \text{si } tab \text{ est un tableau vide} \\ milieu & \text{si } milieu \text{ est l'indice du milieu de } tab \text{ et } tab[milieu] = val \\ rechDicho(tabGauche, val) & \text{si } tab[milieu] > val \text{ où } milieu \text{ est l'indice du milieu de } tab \\ & \quad \text{et } tabGauche \text{ est la moitié gauche de } tab \\ rechDicho(tabDroit, val) & \text{si } tab[milieu] < val \text{ où } milieu \text{ est l'indice du milieu de } tab \\ & \quad \text{et } tabDroit \text{ est la moitié droite de } tab \end{cases}$$

Q1. En remarquant qu'une «moitié» de tableau peut être représentée par ses indices de début et de fin, écrire la fonction récursive `int rechDicho(int[] tab, int val, int idebut, int ifin)` qui recherche la valeur `val` dans la partie du tableau `tab` définie par ses indice de début `idebut` et de fin `ifin`. Votre fonction ne devrait pas utiliser de boucle.

Q2. Est-ce que la fonction écrite est récursive terminale ?

Exercice 3 : Tri rapide

Jusque présent nous avons utilisé la récursivité pour des algorithmes qui peuvent aussi être implémentés en utilisant une boucle. Nous allons voir maintenant un algorithme qui ne peut pas être implémenté sans récusivité [1]

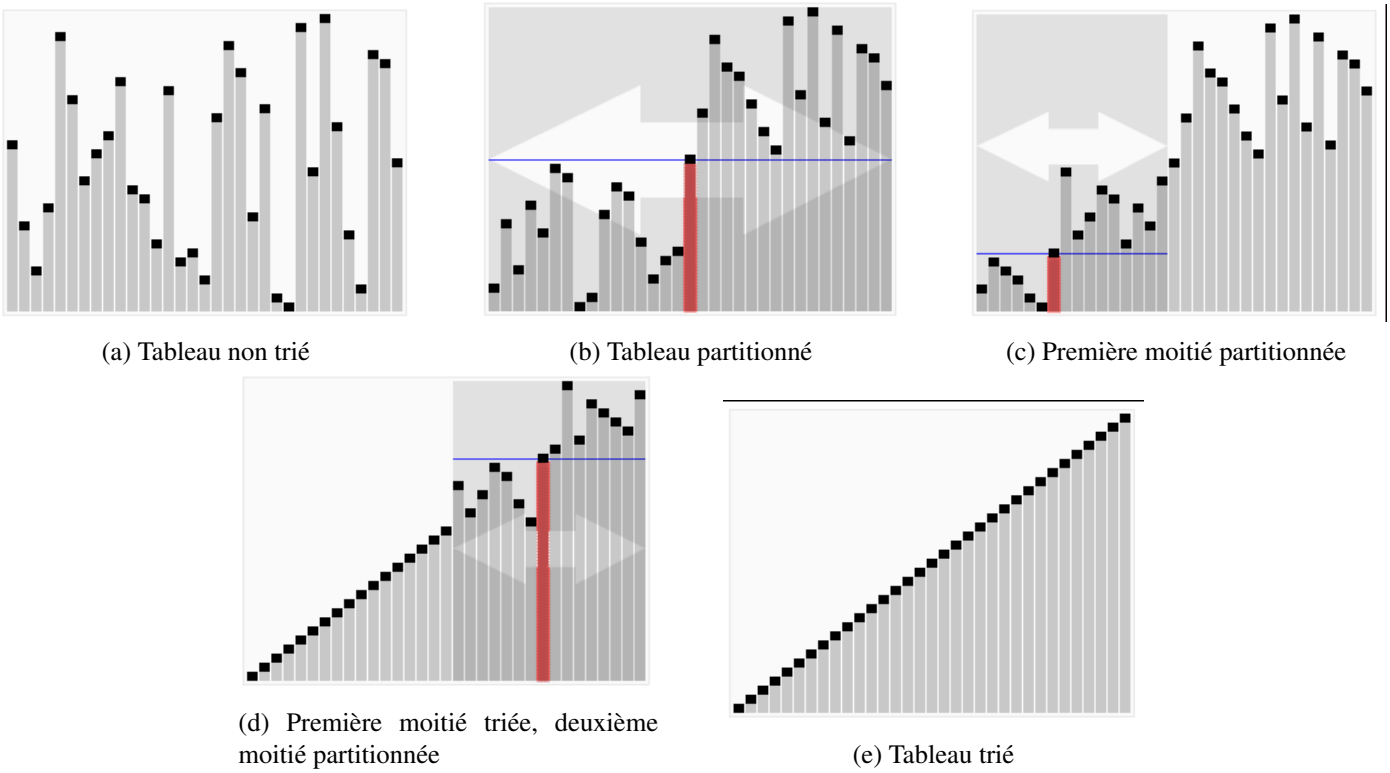
Il s'agit du *tri rapide*. C'est une méthode de tri de tableaux qui, comme son nom l'indique, est en général très efficace.

L'algorithme de tri rapide utilise le principe *diviser pour régner*. Voyons le fonctionnement de l'algorithme sur un exemple. Étant donné le tableau de la Fig. **??** (la grandeur de chaque valeur est représentée par la hauteur du bâton) on commence par *partitionner* le tableau en mettant les petites valeurs au début et les grandes valeurs à la fin. Une valeur est «petite» ou «grande» comparée à une valeur spéciale appelée *pivot*. Puis, on met le pivot entre les petites et les grandes valeurs. On peut voir sur la Fig. **??** le résultat du partitionnement, avec le pivot en rouge au milieu. À gauche toutes les valeurs sont plus petites que le pivot, et à droite toutes les valeurs sont plus grandes que le pivot. L'idée essentielle est que maintenant, si on trie séparément la moitié gauche et la moitié droite, on va obtenir un tableau trié. Pour trier les

1. Ou sans l'utilisation d'une pile, qui est un mécanisme équivalent.

deux moitiés, on précède de la même manière. On choisit un pivot, on partitionne, on met le pivot au milieu comme sur la Fig. **??**, et on recommence avec les nouvelles moitiés d'un côté et de l'autre du pivot.

Sur la Fig. **??** on voit le résultat quand la moitié gauche est déjà triée par ce processus récursif, et on vient de partitionner la moitié droite. Finalement, sur la Fig. **??** on a le résultat final qui est un tableau entièrement trié.



(a) Tableau non trié

(b) Tableau partitionné

(c) Première moitié partitionnée

(d) Première moitié triée, deuxième moitié partitionnée

(e) Tableau trié

FIGURE 1 – Illustration de l'algorithme de tri rapide. Source des images: Wikipedia.

Ainsi, pour implémenter le tri rapide, vous devez écrire deux procédures.

```
void partitionner (int[] tab, int idebut, int ifin, int pivot)
void triRapide (int[] tab, int idebut, int ifin)
```

La procédure `triRapide` est forcément récursive et, qui plus est, ne peut pas être écrite autrement qu'en utilisant la récursion.

La procédure `partitionner` peut s'écrire sans utiliser la récursion, mais pour vous exercer vous aller l'écrire récursivement.

Récupérer sur Moodle le fichier `TriRapide.java` qui contient les signatures de ces procédures, ainsi que des tests.

Q3. La procédure `partitionner` doit partitionner le «morceau» de tableau défini par les indices `idebut` et `ifin`, et doit retourner l'indice de la première valeur plus grande que le pivot. La valeur de retour servira pour définir les moitiés de tableau des appels récursifs. [2] Écrire la procédure `partitionner` en utilisant ces équations de récurrence, où *partitionner* est abrégé par *part*.

$$part(tab, ideb, ifin, pivot) = \begin{cases} ideb & \text{si } ifin < ideb, \text{ càd } tab \text{ est un tableau vide} \\ & \quad \text{peut arriver quand on vient d'échanger} \\ & \quad \text{deux valeurs par le dernier cas} \\ ideb + 1 & \text{si } ifin = ideb \text{ et } tab[ideb] < pivot \\ ideb & \text{si } ifin = ideb \text{ et } tab[ideb] \geq pivot \\ part(tab, ideb{+}1, ifin, pivot) & \text{si } tab[ideb] < pivot \\ part(tab, ideb, ifin{-}1, pivot) & \text{si } tab[ifin] \geq pivot \\ \text{échanger } tab[ideb] \text{ et } tab[ifin], & \\ \quad \text{puis } part(tab, ideb{+}1, ifin{-}1, pivot) & \text{sinon} \end{cases}$$

2. Contrairement au cas de la recherche dichotomique, ici les deux «moitiés» du tableau résultant du partitionnement ne sont pas forcément de tailles égales. Leurs tailles dépendent de la valeur du pivot, voir par exemple la Fig. **??** où il n'y a pas le même nombre de valeurs de part et d'autre du pivot.

Prolongement: l'algorithme de tri rapide

Q4. Le cas de base pour `void triRapide(int[] tab, int idebut, int ifin)` est lorsque la tableau entre les indices `idebut` et `ifin` est vide ou contient une seule case, dans quel cas il est forcément trié. Dans ce cas, la procédure doit simplement retourner.

Pour le cas récusif, `void triRapide(int[] tab, int idebut, int ifin)` doit d'abord choisir une valeur pour le pivot. Celle-ci doit être n'importe quelle valeur présente dans le tableau entre les indices `idebut` et `ifin`. Ainsi vous allez prendre la dernière valeur, càd `pivot = tab[ifin]`.

Ensuite, il faut partitionner la tableau entre `idebut` et `ifin` en utilisant la procédure `partitionner` avec le pivot choisi. Soit `m` l'indice retourné par l'appel `partitionner(tab, idebut, ifin, pivot)`. On échange les valeurs `tab[m]` et `tab[ifin]` de manière à ce que le pivot se trouve entre les petites et les grandes valeurs.

On peut maintenant appeler la procédure `triRapide` sur le sous-tableau à gauche du pivot et sur le sous-tableau à droite du pivot. Rappelons que le pivot a été rangé à l'indice `m`. Chacun des sous-tableaux doit exclure la case à l'indice `m`, qui contient déjà la bonne valeur.

Q5. Vous allez finalement visualiser les appels récursifs à `triRapide`, en utilisant les fonctions fournies à la fin du fichier `TriRapide.java`.

Modifiez votre fonction `triRapide` comme suit:

```
void triRapide(int[] tab, int idebut, int ifin) {
  // Normalement votre fonction commence par le cas de base
  // ressemblant à ceci
  if (fin - debut <= 0) {
    return;
  }

  // Ajouter ici ces instructions
  profondeur++;
  afficher(tab, idebut, ifin);

  // Ici vient le code du cas de récurrence qu'on garde tel quel


  // À la fin de la fonction, ajouter cette instruction
  profondeur--;
}
```

Exécuter maintenant `void algorithm()` (en le renommant au préalable), et expliquer l'affichage produit.