TP n°17 – Tris

1 Tri fusion

On rappelle le principe de ce tri :

- On découpe la liste ou le tableau en deux parties de taille égales (à un élément près)
- On applique récursivement le tri aux deux parties. Si ces parties sont vides ou réduites à 1 élément, elles sont triées
- On fusionne les deux listes/tableaux triés en intercalant les éléments pour qu'ils soient triés.

- **Q1.** Faire un dessin de l'exécution du tri pour les listes [4 ; 3 ; 8 ; 2 ; 7 ; 1 ; 5] et [9 ; 1 ; 5 ; 4 ; 7 ; 2 ; 8 ; 3 ; 6].

Pour diviser une liste de taille n, on peut prendre la première moitié et la seconde moitié ou alors prendre les éléments d'indices pairs et ceux d'indices impairs.

- **Q2.** Quelle méthode est plus facile à appliquer aux listes Ocaml ? Aux tableaux Ocaml ?
- **Q3.** Écrire les fonctions `partition_list : 'a list -> 'a list * 'a list` et `partition_tab : 'a array -> 'a array * 'a array`.

Pour la fusion de deux listes/tableaux l_1 et l_2, il suffit de remplir une nouvelle liste/tableau en rajoutant à chaque fois le plus petit des premiers éléments de l_1 et l_2.

- **Q4.** Écrire la fonction `fusion_list : 'a list * 'a list -> 'a list` qui fusionne des listes triées.
- **Q5.** En déduire la fonction `tri_fusion_list : 'a list -> 'a list` qui trie une liste.

- **Q6.** Écrire la fonction `fusion_tab : 'a array * 'a array -> 'a array`.
- **Q7.** En déduire la fonction `tri_fusion_tab : 'a array -> 'a array` qui trie un tableau.
- **Q8.** La complexité est-elle différente de quand on triait une liste ?

2 Tri rapide

Le tri rapide est un algorithme de tri basé sur le principe de diviser pour régner, comme le tri fusion. Les étapes sont les suivantes :

- On commence par choisir élément du tableau, on le nomme "pivot". On découpe ensuite la liste ou le tableau en deux parties : les éléments < au pivot, et les éléments > aux pivot. Les éléments égaux au pivot peuvent êtré placés n'importe où, cela change selon la méthode de partition.
- On applique récursivement le tri aux deux parties. Si ces parties sont vides ou réduites à 1 élément, elles sont triées
- On concatène simplement les parties triées.

- **Q9.** Reprendre les même listes que pour le tri fusion et dessiner l'exécution du tri rapide, en prenant à chaque fois comme pivot le premier élément de la liste.

On va commencer par implémenter le tri pour des listes.

- **Q10.** Écrire une fonction `partition_pivot_list:'a list -> 'a -> 'a list*'a list*'a list` qui prend en entrée une liste et un pivot et renvoie les listes des éléments plus petits que le pivot, égaux au pivot et plus grands que le pivot.
- **Q11.** En déduire une fonction `tri_rapide_list:'a list -> 'a list` qui effectue le tri rapide. On choisira le premier élément comme pivot (c'est l'élément le moins coûteux à récupérer).

Ici on crée des listes à chaque partition, donc le tri ne s'effectue pas en place.

Ce tri s'effectue pourtant assez naturellement en place : si on réussit à avoir après l'étape de partitionnement les éléments inférieurs au pivot à gauche et ceux supérieurs au pivot à droite, alors l'étape de rassemblement des sous-solutions est triviale.

On va présenter deux algorithmes qui modifient en place le tableau de cette manière pour partitionner. **Le reste de cette partie est en C.**

Méthode de Lomuto

La première technique pour partitionner le tableau compris entre les indices **`debut`** et **`fin`** consiste à :

- Choisir **`tab[fin]`** comme pivot.

- Maintenir une variable `k` initialisée à `debut`. k sera le numéro du premier élément dont on ne sait pas s'il est à sa place.
- Pour `i` allant de `debut` à `fin-1` : si `tab[i]` est inférieur au pivot, échanger les éléments aux indices `k` et `i` et augmenter `k`
- À la fin, on met le pivot à sa place, en échangeant les éléments aux indices `k` et `fin`.

Remarque : Dans cette méthode, les doublons du pivot sont placés à gauche.

- **Q12.** Écrire une fonction `int partition_lomuto(int* tab, int debut, int fin)` qui prend en entrée le tableau, `debut` et `fin` et renvoie la position du pivot à la fin (i.e. l'endroit où on coupe le tableau en 2).
- **Q13.** En déduire la fonction `void tri_rapide_tableau(int* tab, int taille)` qui effectue le tri en place du tableau.

Méthode de Hoare

Une autre méthode pour partitionner le tableau compris entre les indice `debut` et `fin` en place est la suivante :

- Initialiser une variable `i` à `debut+1` et une variable `j` à `fin`. Choisir le pivot comme étant `tab[debut]`.
- tant que $i \leq j$, si `tab[i]` est plus grand que le pivot et `tab[j]` est plus petit que le pivot, on échange les éléments et on augmente `i` et on diminue `j`. Sinon on augmente `i` ou on diminue `j` selon si c'est `tab[i]` ou `tab[j]` qui ne vérifie pas la condition.
- À la fin la frontière est à l'indice j, on peut y placer le pivot.

- **Q14.** Écrire la fonction `int partition_hoare(int* tab, int debut, int fin)` qui effectue le partitionnement selon la méthode de Hoare.
- **Q15.** Faut-il changer quelquechose à la fonction `tri_rapide_tableau` ?

Étude de l'algorithme

On a montré en cours que les appels récursifs sont corrects, mais on avait pas développé la correction de la méthode d'échange des éléments.

- **Q16.** Prouver que les méthodes de Lomuto et Hoare sont correctes.
- **Q17.** On veut déterminer l'impact du choix du pivot sur le nombre de comparaisons réalisées au cours du tri rapide, par une méthode expérimentale.
  On fixe une taille n pour nos tableaux. Au début vous pouvez fixer $n = 15$ mais ensuite on changera.
  - Écrire une fonction générant des tableaux aléatoire sur n. Il est important que votre tableau soit uniformément choisi parmi tous les tableaux possibles.
  - Créer une variable globale `nbcomp` et modifier votre fonction de tri (avec méthode de Hoare, on pourra également regarder celle de Lomuto) pour compter le nombre de comparaisons effectuées.
  - Écrire une fonction qui teste pour plein ($p = 100$ si cela passe sur les ordinateurs) de tableaux et fait la moyenne du nombre de comparaisons à n fixé.
  - Faire le test pour plusieurs valeurs de n et afficher.
    On peut reprendre le code affiche.py du TP 7 qui affiche une courbe si on lui donne un fichier avec des lignes de la forme :

    ```
    abcisse ordonnée\n
    ```

  On pourra également comparer ce qui se passe quand on choisit le pivot comme étant le premier élément, le dernier élément ou un élément d'indice aléatoire. en modifiant le code des deux méthodes en conséquence

Tirer des nombres au hasard en C :

Rajouter `#include <time.h>` en haut du fichier.

Dans le `main`, écrire `srand(time(NULL));`.

Pour tirer un entier entre 0 et $n-1$, écrire `int truc_alea = rand()%n;`