TP Algorithmique et Programmation no 1

Tableaux

Préambule

— Il vous est demandé de *respecter scrupuleusement la spécification* de chacune des fonctions à programmer ; celles-ci vous sont données sous la forme de commentaire java normalisé : vous ferez de même dans vos fichiers. (NB : voir le document « *Préparation des TP de programmation* », partie 2.)
— Les opérations de lecture au terminal se feront *exclusivement avec les fonctions présentées dans le document « Entrées/Sorties en java »* (voir la documentation en ligne `http://etudiant.istic.univ-rennes1.fr/current/esir1/prog/documentation`).

1 Tri, interclassement et recherche

Les fonctions à programmer (sauf les tests unitaires) seront placées dans une classe nommée TriDicho. En outre, vous respecterez les spécifications imposées, y compris les noms de fonction.

1.1 Initialisation et affichage

Programmez les fonctions statiques suivantes :

1. fonction lireTableau : cette fonction place dans un tableau de capacité donnée une suite de nombres entiers lus au clavier ; la saisie se termine quand la valeur *valfin* est saisie ou quand le tableau est plein.
   Remarque : l'entier *valfin* ne doit pas être placé dans le tableau.
   Cette fonction renvoie le nombre de nombres placés dans le tableau.

```
/**
 * initialiser un tableau avec les valeurs d'une suite de nombres entiers lus au clavier
 * la suite est terminée par valfin
 *
 * @param tnb    : tableau de nombres (déjà créé) à initialiser
 * @param valfin : valeur qui met fin à la saisie ; ne doit PAS être placée dans le tableau
 * @param entree : scanner d'entrée où se fait la lecture
 * @post         : le tableau contient N nombres entiers (0 <= N <= tnb.length)
 * @return       : nombre de nombres placés dans le tableau (N)
 */
static int lireTableau(int [ ] tnb, int valfin, Scanner entree);
```

2. fonction afficherTableau : affiche les premiers éléments d'un tableau.

```
/**
 * afficher les nb premiers éléments d'un tableau.
 * @param tnb : tableau initialisé
 * @param nb  : nombre d'éléments en tête du tableau
 * @pre 0 <= nb <= tnb.length
 * @post le tableau n'est pas modifié
 */
static void afficherTableau(int [ ] tnb, int nb);
```

1.2 Test unitaire

Consultez le document « *Test unitaire avec JUnit* » (voir aussi `http://etudiant.istic.univ-rennes1.fr/current/esir1/prog/documentation`).

Copiez le fichier TestUnitaireLecture.java dans le répertoire de votre projet. C'est un programme de test unitaire que vous devez utiliser pour tester la fonction lireTableau.

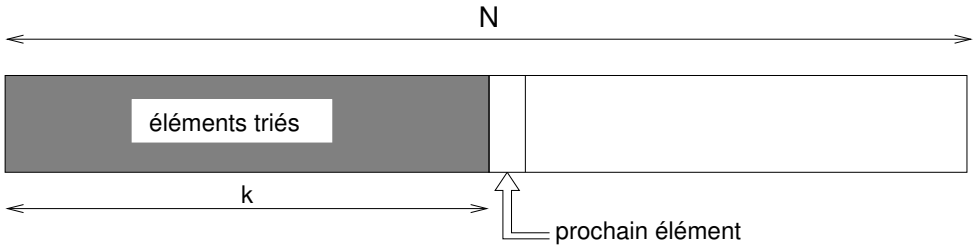
Si vous rencontrez des erreurs lorsque vous compilez le programme de test, vérifiez d'abord si vous avez fait ce qui est indiqué dans le document ci-dessus ; vérifiez ensuite que votre fonction respecte bien la spécification imposée.

Si l'un des tests échoue à cause d'un temps d'exécution trop long, essayez de doubler la durée autorisée pour ce test. S'il échoue encore, c'est peut-être que votre algorithme est trop compliqué...

Votre fonction est considérée comme testée (mais pas obligatoirement correcte), si elle passe avec succès tous les tests.

1.3 Tri par insertion

On cherche à trier en ordre croissant les N premiers éléments d'un tableau ; pour celà, on procède ainsi :
- on trie progressivement les éléments du tableau de la gauche vers la droite *en suivant le principe présenté plus bas* ;
- au bout d'un certain nombre d'itérations, on a trié les k premiers éléments et on obtient la situation du schéma ci-dessous :



Principe : il s'agit maintenant d'*insérer* le premier élément non trié « à sa place » parmi les k premiers éléments (déjà triés) pour obtenir un tableau dont les k+1 premiers éléments sont triés ; pour minimiser le nombre d'affectations, *on évitera de procéder par permutation d'éléments adjacents.*
Une fois cet élément placé, on se retrouve dans la situation précédente et il suffit d'appliquer le même principe aux éléments restants.

1. Programmez la fonction trierInsertion qui effectue le tri des N premiers éléments d'un tableau de nombres entiers *selon l'algorithme ci-dessus* ; la signature de la fonction peut se déduire du programme de test.
2. Testez votre algorithme de tri avec le programme de test unitaire TestUnitaireTriInsertion.java.

Questions :

1. Est-ce que le programme de test permet de garantir que votre fonction effectue un tri des éléments d'un tableau ? Expliquez.
2. Expliquez pourquoi le programme de test ne permet pas de garantir que votre fonction réalise ce tri avec l'algorithme demandé.

1.4 Interclassement de deux tableaux triés

1. Programmez une fonction qui, étant donné deux tableaux triés de nombres entiers t1 et t2, crée et renvoie un *nouveau tableau* trié t3 obtenu par interclassement des nb1 premiers éléments de t1 et des nb2 premiers éléments de t2 : le nombre d'éléments de t3 sera donc nb1+nb2 et contiendra tous les éléments de t1 et tous les éléments de t2.

   Exemple : $t1 = \{-5, 7, 13\}$, $t2 = \{-10, -5, 7, 10\} \Rightarrow t3 = \{-10, -5, -5, 7, 7, 10, 13\}$

   NB1 : il ne s'agit *pas* de copier les éléments de t1 puis ceux de t2 dans t3 puis de trier t3.

   NB2 : il ne s'agit *pas non plus* de copier les éléments de t1 dans t3 puis d'insérer ceux de t2 dans t3.
2. Testez votre fonction d'interclassement avec le programme TestUnitaireInterclassement.java.

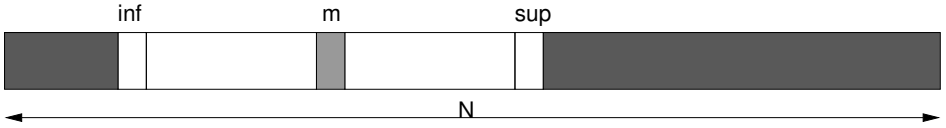
1.5 Recherche dichotomique

Remarque : La recherche par *dichotomie* d'un élément dans un tableau *trié* est un algorithme très efficace : pour un tableau d'un million d'éléments, cet algorithme permet de déterminer en 20 itérations au maximum la présence ou l'absence de l'élément cherché, alors qu'une recherche séquentielle aurait besoin de 500 000 itérations en moyenne...

principe général de la recherche par *dichotomie* d'un élément e dans un tableau T *trié* de N éléments :

1. regarder si l'élément cherché est au milieu du tableau ; si oui, l'élément cherché est trouvé ; si non :
   — *découper le tableau en deux moitiés ;*
   — poursuivre la recherche de l'élément dans une moitié en *ignorant la partie du tableau où l'élément ne peut en aucun cas se trouver* ;
2. On applique alors le *même principe* sur la moitié où se poursuit la recherche.

détail de la progression : On suppose qu'on a déjà effectué un certain nombre d'itérations selon le principe ci-dessus *sans avoir trouvé l'élément cherché* ; la recherche se limite donc au sous-tableau compris entre les indices inf et sup inclus ($[inf \cdots sup]$) ;



soit m l'indice de l'élément situé « *au milieu* » du sous-tableau $[inf \cdots sup]$:

1. si $e = T[m]$ on a trouvé l'élément cherché ;
2. si $e < T[m]$
   — que peut-on conclure des éléments $T[j]$ tels que $m \leq j < N$?
   — on poursuit la recherche de e *par dichotomie* dans une des deux « *moitiés* » du sous-tableau $[inf \cdots sup]$; indiquez précisément les bornes de cette « *moitié* » et **expliquez pourquoi il est possible d'ignorer $T[m]$ dans la suite de la recherche** ;
3. si $e > T[m]$
   — que peut-on conclure des éléments $T[i]$ tels que $0 \leq i \leq m$?
   — on poursuit la recherche de e *par dichotomie* dans une des deux « *moitiés* » du sous-tableau $[inf \cdots sup]$; indiquez précisément les bornes de cette « *moitié* » et **expliquez pourquoi il est possible d'ignorer $T[m]$ dans la suite de la recherche** ;

cas d'arrêt : la recherche s'arrête quand le sous-tableau compris entre les indices inf et sup inclus est *vide* ou quand l'élément cherché est *trouvé*.

Réalisation

1. Programmez la fonction rechercheDichotomique qui effectue la recherche dichotomique d'un nombre parmi les N premiers éléments d'un tableau en respectant *scrupuleusement* l'analyse ci-dessus ; le résultat sera un indice i tel que $T[i] = e$ si l'élément e est présent dans le tableau ; -1 si l'élément est absent du tableau ; la signature de la fonction peut se déduire du programme de test.
2. Testez votre fonction de recherche avec le programme TestUnitaireRechercheDichotomique.java. Si l'un des tests échoue à cause d'un temps d'exécution trop long, essayez de doubler la durée autorisée pour ce test. S'il échoue encore, c'est peut-être que vous n'avez pas respecté l'algorithme ci-dessus.