TP R1.01.P2 – Semaines 47 - 48 - 49

Présentation du TP

Ce TP consiste à développer une nouvelle classe *TrisTableau* qui contiendra le codage d'opérations plus complexe sur un tableau d'entiers.

Cette classe sera codée sur 3 semaines et sera rendue en fin de semaine 49 (**samedi 07/12 à 23h55 au + tard**). **Attention**, un **malus** sera appliqué sur la note de TP : -2 points pour tout retard de rendu (même d'une minute !). Le nommage du fichier sera impérativement *Nom_Prenom.zip* (PAS de fichier .rar).

Les méthodes de la classe *SimplesTableau* peuvent être **réutilisées** (voir page 3) dès que vous le jugez utile (notamment par exemple, les méthodes de remplissage, aléatoire ou non, de tableaux).

Dans ce développement de la classe *TrisTableau*, on vous demande de coder des méthodes de recherche (séquentielle et dichotomique), des méthodes de tris (tri simple, tri rapide etc.) et d'évaluer leur efficacité.

L'ordre de développement des méthodes est le suivant :

1. *rechercheSeq*
2. *verifTri*
3. *rechercheDicho (attention, recherche sur un tableau trié !)*
4. *triSimple*
5. *separer*
6. *triRapideRec*
7. *triRapide*
8. *creerTabFreq*
9. *triParComptageFreq*
10. *triABulles*

Le détail du rôle de chaque méthode, de même que le détail des paramètres d'entrée et de sortie est présenté dans le document en annexe intitulé *javaDoc de la classe TrisTableau*.

Le test des méthodes et test de l'efficacité

Les 10 méthodes ci-dessus feront l'objet comme précédemment d'un test unitaire qui met à l'épreuve la méthode dans tous les cas de figures : cas normal, cas limites et cas d'erreurs. Ces 10 méthodes de test se nommeront *testRechercheSeq*, *testRechercheDicho, testVeriftri etc*.

Concernant la mesure de l'**efficacité** des algorithmes de recherche et de tri, on écrira les 6 méthodes particulières :

- *void testRechercheSeqEfficacite()* $\Theta(n)$
- *void testRechercheDichoEfficacite()* $\Theta(log_2 n)$
- *void testTriSimpleEfficacite()* $\Theta(n^2)$
- *void testTriRapideEfficacite()* $\Theta(n log_2 n)$
- *void testTriParComptageFreqEfficacite()* $\Theta(??)$
- *void testTriABullesEfficacite()* $\Theta(??)$

Pour chacune des 6 méthodes (2 de *recherche* et 4 de *tri*), on veut cette fois les tester sur des tableaux de très grandes tailles (n = 100.000 valeurs et au-delà) pour mettre en évidence l'efficacité en Θ des algorithmes (ne PAS afficher les tableaux d'une telle taille !). Deux indicateurs nous serviront à évaluer l'efficacité :

- *cpt* : un compteur d'opérations élémentaires. Celui-ci sera déclaré en variable globale de type *long* (*64 bits*) et sera initialisé à zéro avant l'exécution de la méthode. Ce compteur sera incrémenté de 1 à chaque passage dans la boucle la + imbriquée. De la sorte on fait une approximation du nombre total d'opérations en considérant que pour la recherche séquentielle $f(n) \approx n$, pour la recherche dichotomique $f(n) \approx log_2 n$, pour le tri simple $f(n) \approx n^2/2$ et pour le tri rapide $f(n) \approx n log_2 n$.

  En fin d'exécution de l'algorithme, on doit observer que $cpt/n = cste1$ pour la recherche séquentielle, $cpt/log_2 n = cste2$ pour la recherche dichotomique, $cpt/n^2 = cste3$ pour le tri simple et $cpt/nlog_2 n = cste4$ pour le tri rapide (à vous de voir pour les 2 autres tris).

  En reprenant votre cours R1.01.P2 dans lequel on vous donne (dans le cas d'algorithmes bien précis !) le nombre exact d'opérations *f(n)* pour les recherches séquentielle et dichotomique et pour les tris simple et rapide, calculez de manière théorique les constantes *cste1*, *cste2* et *cste3* (i.e. $lim_{n->\infty} f(n)/\Theta$ où $\Theta = n, log_2 n, n^2$) suivant le type de recherche ou de tri (**on ne calculera pas de manière théorique *cste4* qui est + complexe à obtenir**). Ces constantes théoriques doivent apparaître dans votre tableau comparatif (voir dernier paragraphe) et doivent être comparées aux mesures empiriques cpt/Θ : en principe ça doit être la même valeur, mais il faut le vérifier !

- *tps* : le temps d'exécution de la méthode de recherche en millisecondes ou nanosecondes (méthode *System.currentTimeMillis()* ou *System.nanoTime()* de *Java*). Bien faire le différentiel du temps mesuré avant et après l'exécution de la méthode.

  Que doit-on observer :

  - Pour la recherche séquentielle, multiplier *n* (nombre d'éléments dans le tableau) par *k* doit multiplier le temps d'exécution par *k*.

  - Pour la recherche dichotomique, multiplier *n* par 2^k doit augmenter le temps d'exécution de *k* (en réalité $k * T$ secondes où $T = 1/f$, *f* étant la fréquence de l'horloge)

    $log_2(n\ 2^k) = log_2 n + log_2 2^k = log_2 n + k$

    Ceci signifie que si *n* est élevé au départ (2^{15} par exemple) le temps d'exécution n'augmentera quasiment pas sauf si *k* devient significatif par rapport à 15 !

  - Pour le tri simple, multiplier *n* (nombre d'éléments dans le tableau) par *k* doit multiplier le temps d'exécution par k^2.

  - Pour le tri rapide, multiplier *n* par 2^k doit augmenter le temps d'exécution par une valeur calculable :

    $(n\ 2^k) log_2(n\ 2^k) = n\ 2^k\ (\ log_2 n + k\ )$

    Ceci signifie que si *n* est élevé au départ (2^{15} par exemple) le temps d'exécution augmentera proportionnellement à 2^k sauf si *k* devient significatif par rapport à 15 !

Attention à s'y prendre correctement :

- Pour la recherche séquentielle, *nbElem* = 1.000.000 au départ est faisable (sauf s'il y a dépassement de la capacité mémoire de votre *PC*). Il faudra évaluer l'efficacité dans le pire des cas (pour être

proportionnel à *nbElem*) et donc placer l'élément à rechercher en fin de tableau (*nbElem-1*), ou rechercher une valeur qui n'est pas présente dans le tableau.

- Pour la recherche dichotomique, votre tableau de départ **doit être trié**. Pour trier ce tableau, remplir un tableau de *nbElem* valeurs, ces valeurs allant de *zéro* à *nbElem-1*, qui sera forcément trié. Aller au-delà de *nbElem* = 1.000.000 est tout à fait faisable (sauf s'il y a dépassement de la capacité mémoire de votre *PC*). Dans certains cas (tableau trop petit) utiliser *System.nanoTime()* pour la mesure du temps car la milliseconde ne suffira pas.

- Pour le tri simple, la méthode la plus coûteuse en temps de calcul, *nbElem* = 15.000 au départ est faisable, mais *nbElem* = 500.000 sera déjà un maximum.

D'une manière générale, pensez à utiliser des types de données de grandes capacités : *long* pour les entiers et *double* pour les réels. Insérez l'instruction *cpt++* dans votre code des algorithmes de tri à l'intérieur des boucles les + imbriquées, sinon le rapport cpt/Θ n'aura aucune signification. N'oubliez pas que la mesure du temps de calcul est approximative et que cette mesure dépend, notamment, de la charge de la machine au moment où vous exécutez l'algorithme.

Rappel d'une formule du logarithme : $\log_n A = \log_{10} A / \log_{10} n$ (utiliser *Math.log10 (double d)* de Java qui renvoie un type *double*).

Enfin, il est possible d'utiliser directement les méthodes de la classe *SimplesTableau* (sans faire de copier/coller de code) dans votre classe *TrisTableau* :

- après avoir récupéré la classe compilée *SimplesTableau.class* disponible sur *Moodle R1.01*, placer cette classe dans le répertoire */class* de votre arborescence,
- déclarez une variable globale *monSpleTab* de type *SimplesTableau* dans votre classe *TrisTableau*, de la manière suivante :

```
class TrisTableau {

    SimplesTableau monSpleTab = new SimplesTableau();

    void principal() {…}

    …
}
```

Exemple d'utilisation de la méthode afficherTab(…) de la classe SimplesTableau : `monSpleTab.afficherTab(…);`

Tableau comparatif

Dans un document intitulé *Comparatifs.pdf*, effectuez un comparatif des différentes méthodes de tris (+ recherche dichotomique) suivant différentes valeurs de « n » (taille du tableau initial) et suivant les différentes données obtenues en exécutant les méthodes *test…Efficacite()*.

Les résultats obtenus doivent être commentés en quelques lignes et le détail des développements mathématiques qui permet d'aboutir aux constantes (*cste1*, *cste2*, *cste3*) doit également être donné. Exemple de tableau type :

Type de méthode	**n**	**Θ**	**cpt / Θ**	**$lim_{n->\infty}$ f(n) / Θ cste1, cste2, cste3**	**temps (millisecondes)**
Recherche séquentielle					
	…				
	…				
Recherche dichotomique					
	…				
	…				
Tri simple					
	…				
	…				
Tri rapide					
	…				
	…				
Tri comptage de fréquences					
	…				
	…				
Tri à bulles					
	…				
	…				

Rendu

Votre rendu sera une archive *Nom_Prenom.zip* (pas de *.rar*) qui contient :

- Le **source** de la classe *TrisTableau* (*TrisTableau.java*).
- Le fichier *Comparatifs.pdf* qui contiendra également les développements mathématiques de calcul des constantes *cste1*, *cste2*, *cste3*.

ANNEXE : JavaDoc de la classe TrisTableau

Class TrisTableau

java.lang.Object
TrisTableau

public class **TrisTableau**
extends java.lang.Object

Cette classe effectue des opérations plus complexes sur un tableaux d'entiers : recherche dichotomique, tris, etc. La taille d'un tableau est par définition le nombre TOTAL de cases = tab.length. Un tableau d'entiers créé possède nbElem élements qui ne correspond pas forcément à la taille du tableau. Il faut donc toujours considéré que nbElem <= tab.length (= taille).

Il est fait usage de la classe SimplesTableau pour accéder aux méthodes de cette classe.

Author:

J-F. Kamp - septembre 2024

Field Summary

Fields

Modifier and Type	Field	Description
(package private) long	cpt	Variable globale, compteur du nombre d'opérations
(package private) SimplesTableau	monSpleTab	Variable globale permettant l'accès aux méthodes de la classe SimplesTableau

Constructor Summary

Constructors

Constructor	Description
TrisTableau()	

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
(package private) int[]	creerTabFreq (int[] leTab, int nbElem)	A partir d'un tableau initial passé en paramètre "leTab", cette méthode renvoie un nouveau tableau "tabFreq" d'entiers où chaque case contient la fréquence d'apparition des valeurs dans le tableau initial.
(package private) void	echange(int[] leTab, int ind1, int ind2)	Echange les contenus des cases du tableau passé en paramètre, cases identifiées par les indices ind1 et ind2.
(package private) void	principal()	Le point d'entrée du programme.
(package private) int	rechercheDicho (int[] leTab, int nbElem, int aRech)	Recherche dichotomique d'une valeur dans un tableau.
(package private) int	rechercheSeq (int[] leTab, int nbElem, int aRech)	Recherche séquentielle d'une valeur dans un tableau.
(package private) int	separer(int[] tab, int indL, int indR)	Cette méthode renvoie l'indice de séparation du tableau en 2 parties par placement du pivot à la bonne case.
(package	triABulles	Tri par ordre croissant d'un tableau selon la méthode du tri à bulles : tant que le tableau

private) void	(int[] leTab, int nbElem)	qu'il reste à trier a au moins 2 cases, permuter le contenu de 2 cases successives si leTab[i] > leTab[i+1].
(package private) void	**triParComptageFreq** (int[] leTab, int nbElem)	Tri par ordre croissant d'un tableau selon la méthode du tri par comptage de fréquences.
(package private) void	**triRapide**(int[] leTab, int nbElem)	Tri par ordre croissant d'un tableau selon la méthode du tri rapide (QuickSort).
(package private) void	**triRapideRec** (int[] tab, int indL, int indR)	Méthode de tri récursive selon le principe de séparation.
(package private) void	**triSimple**(int[] leTab, int nbElem)	Tri par ordre croissant d'un tableau selon la méthode simple : l'élément minimum est placé en début de tableau (efficacité en n carré).
(package private) boolean	**verifTri**(int[] leTab, int nbElem)	Vérifie que le tableau passé en paramètre est trié par ordre croissant des valeurs.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

cpt

`long cpt`

Variable globale, compteur du nombre d'opérations

monSpleTab

`SimplesTableau monSpleTab`

Variable globale permettant l'accès aux méthodes de la classe SimplesTableau

Constructor Detail

TrisTableau

`public TrisTableau()`

Method Detail

principal

`void principal()`

Le point d'entrée du programme.

rechercheSeq

`int rechercheSeq(int[] leTab, int nbElem, int aRech)`

Recherche séquentielle d'une valeur dans un tableau. La valeur à rechercher peut exister en plusieurs exemplaires mais la recherche s'arrête dès qu'une première valeur est trouvée. On suppose que le tableau passé en paramètre est créé et possède au moins une valeur (nbElem > 0). Ceci ne doit donc pas être vérifié.

Parameters:

`leTab` - le tableau dans lequel effectuer la recherche

`nbElem` - le nombre d'entiers que contient le tableau

`aRech` - l'entier à rechercher dans le tableau

Returns:

l'indice (>=0) de la position de l'entier dans le tableau ou -1 s'il n'est pas présent

triRapide

`void triRapide(int[] leTab, int nbElem)`

Tri par ordre croissant d'un tableau selon la méthode du tri rapide (QuickSort). On suppose que le tableau passé en paramètre est créé et possède au moins une valeur (nbElem > 0). Ceci ne doit donc pas être vérifié. Cette méthode appelle triRapideRec(...) qui elle effectue réellement le tri rapide selon la méthode de séparation récursive.

Parameters:

`leTab` - le tableau à trier par ordre croissant

`nbElem` - le nombre d'entiers que contient le tableau

triRapideRec

`void triRapideRec(int[] tab, int indL, int indR)`

Méthode de tri récursive selon le principe de séparation. La méthode s'appelle elle-même sur les tableaux gauche et droite par rapport à un pivot.

Parameters:

`tab` - le tableau sur lequel est effectué la séparation

`indL` - l'indice gauche de début de tableau

`indR` - l'indice droite de fin de tableau

separer

`int separer(int[] tab, int indL, int indR)`

Cette méthode renvoie l'indice de séparation du tableau en 2 parties par placement du pivot à la bonne case.

Parameters:

`tab` - le tableau des valeurs

`indR` - indice Right de fin de tableau

`indL` - indice Left de début de tableau

Returns:

l'indice de séparation du tableau

triABulles

`void triABulles(int[] leTab, int nbElem)`

Tri par ordre croissant d'un tableau selon la méthode du tri à bulles : tant que le tableau qu'il reste à trier a au moins 2 cases, permuter le contenu de 2 cases successives si leTab[i] > leTab[i+1]. On suppose que le tableau passé en paramètre est créé et possède au moins une valeur (nbElem > 0). Ceci ne doit donc pas être vérifié.

Parameters:

`leTab` - le tableau à trier par ordre croissant

`nbElem` - le nombre d'entiers que contient le tableau

triSimple

```
void triSimple(int[] leTab, int nbElem)
```

Tri par ordre croissant d'un tableau selon la méthode simple : l'élément minimum est placé en début de tableau (efficacité en n carré). On suppose que le tableau passé en paramètre est créé et possède au moins une valeur (nbElem > 0). Ceci ne doit donc pas être vérifié.

Parameters:

`leTab` - le tableau à trier par ordre croissant

`nbElem` - le nombre d'entiers que contient le tableau

verifTri

```
boolean verifTri(int[] leTab, int nbElem)
```

Vérifie que le tableau passé en paramètre est trié par ordre croissant des valeurs. On suppose que le tableau passé en paramètre est créé et possède au moins une valeur (nbElem > 0). Ceci ne doit donc pas être vérifié.

Parameters:

`leTab` - le tableau à vérifier (trié en ordre croissant)

`nbElem` - le nombre d'entiers présents dans le tableau

Returns:

true si le tableau est trié

echange

```
void echange(int[] leTab, int ind1, int ind2)
```

Echange les contenus des cases du tableau passé en paramètre, cases identifiées par les indices ind1 et ind2.

Parameters:

`leTab` - le tableau

`ind1` - numéro de la première case à échanger

`ind2` - numéro de la deuxième case à échanger

triParComptageFreq

```
void triParComptageFreq(int[] leTab, int nbElem)
```

Tri par ordre croissant d'un tableau selon la méthode du tri par comptage de fréquences. On suppose que le tableau passé en paramètre est créé et possède au moins une valeur (nbElem > 0). Ceci ne doit donc pas être vérifié.

Parameters:

`leTab` - le tableau à trier par ordre croissant

`nbElem` - le nombre d'entiers que contient le tableau

creerTabFreq

```
int[] creerTabFreq(int[] leTab, int nbElem)
```

A partir d'un tableau initial passé en paramètre "leTab", cette méthode renvoie un nouveau tableau "tabFreq" d'entiers où chaque case contient la fréquence d'apparition des valeurs dans le tableau initial. Pour simplifier, on suppose que le tableau initial ne contient que des entiers compris entre 0 et max (>0). Dès lors le tableau "tabFreq" se compose de (max+1) cases et chaque case "i" (0<=i<=max) contient le nombre de fois que l'entier "i" apparait dans le tableau initial. On suppose que le tableau passé en paramètre est créé et possède au moins une valeur (nbElem > 0). Ceci ne doit donc pas être vérifié. Par contre, on vérifiera que le min est >= 0. Dans le cas contraire, renvoyer un tableau "null".

Parameters:

`leTab` - le tableau initial

`nbElem` - le nombre d'entiers présents dans le tableau

Returns:

le tableau des fréquences de taille (max+1) ou null si la méthode ne s'applique pas

rechercheDicho

```
int rechercheDicho(int[] leTab, int nbElem, int aRech)
```

Recherche dichotomique d'une valeur dans un tableau. On suppose que le tableau est trié par ordre croissant. La valeur à rechercher peut exister en plusieurs exemplaires, dans ce cas, c'est la valeur à l'indice le + faible qui sera trouvé. On suppose également que le tableau passé en paramètre est créé et possède au moins une valeur (nbElem > 0). Ceci ne doit donc pas être vérifié.

Parameters:

`leTab` - le tableau trié par ordre croissant dans lequel effectuer la recherche

`nbElem` - le nombre d'entiers que contient le tableau

`aRech` - l'entier à rechercher dans le tableau

Returns:

l'indice (>=0) de la position de l'entier dans le tableau ou -1 s'il n'est pas présent