TP n°4

Algorithmes de tri

L'objet de ce TP est de programmer et de comparer l'efficacité de deux algorithmes de tris vu en cours : le tri bulle et le tri rapide. L'entrée de chacun de ces algorithmes est un tableau d'entiers en désordre, et le but est de trier ce tableau par ordre croissant en modifiant l'ordre de ses éléments (c'est-à-dire en effectuant des échanges entre ses cases). Pour chacun de ces algorithmes, la méthode correspondante fera afficher le nombre de comparaisons effectuées, le nombre d'échanges effectués, et le temps d'exécution.

1 Préliminaires

Exercice 1 Dans tout le TP, on utilise des tableaux d'entiers sur lesquels on a le droit d'effectuer quatre opérations élémentaires : accéder à la taille du tableau ; comparer les éléments en position `i` et `j` ; échanger les éléments en position `i` et `j` ; afficher le tableau.

1. Écrire une classe `Tableau` avec un attribut `private int[] t`.
2. Écrire un constructeur copiant dans `t` un tableau passé en argument (on fera bien attention à ne pas copier la référence du tableau, mais bien à faire une copie complète du tableau passé en argument ; sinon, ça ne sert à rien de mettre votre attribut `t` en privé).
3. Écrire quatre méthodes
   - `public int taille()` qui renvoie la taille de `t` ;
   - `public boolean comparer(int i, int j)` qui renvoie `true` si `t[i]` $\leq$ `t[j]` et `false` sinon ;
   - `public void echanger(int i, int j)` qui échange les éléments `t[i]` et `t[j]` du tableau `t` ;
   - `public void afficher()` qui affiche les éléments du tableau.

Dans toute la suite, on ne se servira que de cette classe pour programmer nos algorithmes de tri. En particulier, vous ne pourrez pas utiliser autre chose que nos quatre opérations élémentaires.

2 Deux algorithmes de tri

Exercice 2 ***Tri Bulle***

Le principe du tri bulle est de pousser à droite les plus grands éléments du tableau (de la même manière que les grosses bulles remontent à la surface dans un liquide). À chaque étape, on compare deux éléments consécutifs du tableau, et on les échange si ils sont à l'envers (c'est-à-dire si celui de gauche est plus grand que celui de droite). On commence avec `t[0]` et `t[1]`, puis on continue avec `t[1]` et `t[2]`, puis `t[2]` et `t[3]`, etc. Une fois qu'on a parcouru tout le tableau, on est sûr que le plus grand élément est à la fin du tableau et on recommence.

1. Appliquer le tri bulle à la main sur le tableau | 5 | 3 | 4 | 1 | 2 |. Combien a-t-on effectué de comparaisons et d'échanges ?
2. En général, combien fait-on de comparaisons et d'échanges dans le pire cas et dans le meilleur cas ?
3. Quel est l'invariant de boucle qui permet d'assurer que l'algorithme termine et renvoie bien un tableau trié ?
4. Écrire une méthode `triBulle(Tableau t)` qui effectue le tri bulle. À la fin de l'algorithme, votre méthode doit afficher le nombre de comparaisons effectuées, le nombre d'échanges effectués, et le temps d'exécution. Vérifier sur le tableau | 5 | 3 | 4 | 1 | 2 |.

Exercice 3 ***Tri rapide (Quicksort)***

Le principe du tri rapide est de choisir un élément du tableau (appelé pivot) et de le placer immédiatement à sa place définitive en permutant les éléments de sorte que tous ceux qui sont plus petits que le pivot soient situés à sa gauche, et tous ceux qui sont plus grands que le pivot soient situés à sa droite. Cette étape s'appelle le partitionement du tableau. Ensuite, on recommence l'opération récursivement sur les sous-tableaux situés avant et après le pivot jusqu'à ce que le tableau soit entièrement trié.

1. Appliquer le tri rapide à la main sur le tableau | 7 | 5 | 3 | 6 | 4 | 1 | 8 | 2 |. Combien a-t-on effectué de comparaisons et d'échanges ?
2. En général, combien fait-on de comparaisons et d'échanges dans le pire cas et dans le meilleur cas ?
3. Quel est l'invariant de boucle qui permet d'assurer que l'algorithme termine et renvoie bien un tableau trié ?
4. Écrire une methode `partitionement(Tableau t, int i, int j, int k)` qui prend en argument un tableau `t`, les indices `i` de commencement et `j` de fin d'un sous-tableau, et l'indice `k` du pivot (avec bien sûr $\texttt{i} \leq \texttt{k} \leq \texttt{j}$), et qui partitionne le sous-tableau : elle doit échanger des éléments de `t` de sorte que tous les éléments du sous-tableau entre `i` et `j` qui sont plus petits que `t[k]` se placent avant lui, tandis que tous ceux qui sont plus grands que `t[k]` se placent après lui.
5. Écrire une méthode récursive `triRapideRec(Tableau t, int i, int j)` qui effectue le tri rapide sur la portion du tableau située entre les indices `i` et `j`.
6. Écrire une méthode `triRapide(Tableau t)` qui effectue le tri rapide. Vérifier sur le tableau | 5 | 3 | 4 | 1 | 2 |.

3 Comparaison des tris

Pour comparer l'efficacité de nos deux algorithmes de tri, on peut comparer leur complexité (nombre de comparaison et nombre d'échanges) dans leur pire et leur meilleur cas. Mais quel est l'algorithme le plus rapide si on prend un tableau "au hasard" ? C'est ce qu'on appelle la complexité en moyenne.

Comme nos algorithmes ne dépendent que de l'ordre des éléments dans le tableau, on peut supposer que nos tableaux sont en fait des permutations de $\{1, 2, \ldots, n\}$. On aimerait savoir la complexité de nos algorithmes sur une permutation tirée au hasard. On peut bien sûr la calculer théoriquement, mais on va ici le faire expérimentalement.

Exercice 4 ***Installation de compteurs***

1. Créer une classe `Compteur` ayant deux attributs `nombreComparaisons` et `nombreEchanges` de type `long`. Créer une méthode `somme` qui permet d'ajouter un compteur à un autre.
2. Dans vos deux algorithmes de tri, utiliser cette classe `Compteur` pour renvoyer plutôt qu'afficher le nombre de comparaisons et le nombre d'échanges effectués au cours de l'algorithme.

Exercice 5 ***Génération exhaustive des permutations***

Notre première méthode consiste à générer toutes les permutations de `n` éléments (disons par exemple avec `n` = 10), à faire tourner nos algorithmes sur ces permutations, et à calculer la moyenne de leur nombre de comparaisons et de leur nombre d'échanges.

Pour générer toutes les permutations, on va utiliser l'ordre lexicographique (comme dans le dictionnaire) : pour deux tableaux `p` et `q` qui sont des permutations de $\{1, 2, \ldots, n\}$, on définit :

$$\texttt{p} <_{\text{lex}} \texttt{q} \iff \texttt{p[0]} < \texttt{q[0]}, \text{ ou } \texttt{p[0]} = \texttt{q[0]} \text{ et } \texttt{p[1]} < \texttt{q[1]}, \text{ etc.}$$

1. Quelle est la plus petite permutation pour l'ordre lexicographique $<_{\text{lex}}$? et la plus grande ?

Pour obtenir toutes les permutations, on les génère dans l'ordre lexicographique : on commence par la permutation | 1 | 2 | ... | n | et on calcule son successeur dans l'ordre lexicographique, puis à nouveau son successeur, et ainsi de suite jusqu'à arriver à la permutation | n | n-1 | ... | 1 |.

Pour celà, on doit savoir calculer le successeur d'une permutation `p`. Si `p` n'est pas la permutation finale | n | n-1 | ... | 1 |, on cherche le dernier indice `i` tel que `p[i]` $<$ `p[i+1]`, puis l'indice `j` du plus petit élément qui est plus grand que `p[i]` et qui est situé après lui. Le successeur de `p` dans l'ordre lexicographique est alors obtenu en échangeant `p[i]` avec `p[j]` puis en triant le sous-tableau situé après l'indice `i+1` (inclus).

2. Écrire une méthode `successeur(Tableau p)` qui transforme la permutation `p` en son successeur dans l'ordre lexicographique. Noter que l'on ne renvoie pas un tableau, on transforme simplement le tableau `p`. On utilisera la méthode `triRapideRec` pour trier un sous-tableau.
3. Écrire une méthode `generationPermutations(int n)` qui affiche toutes les permutations de taille `n`. Essayer avec `n` = 4. Combien en trouvez-vous ?
4. Écrire une méthode `complexiteMoyenne(int n)` qui calcule le nombre moyen de comparaisons et d'échanges effectués par nos deux algorithmes de tri lorsqu'ils trient les permutations de taille `n`. Essayer avec `n` = 10.
5. Quel semble être l'algorithme de tri le plus efficace entre le tri bulle et le tri rapide ?
6. Maintenant que l'on sait générer toutes les permutations de taille 6, vérifier quels sont les pire cas et meilleur cas de nos deux algorithmes de tri lorsqu'ils trient un tableau de taille 6.

Exercice 6 ***Génération aléatoire de permutations***

Dans l'exercice précédent, on a vu comment générer toutes les permutations pour calculer la complexité en moyenne de nos algorithmes de tri. Comme le nombre de permutations augmente très rapidement, on ne peut pas aller très loin avec cette méthode. Pour estimer la complexité moyenne de nos algorithmes de tri sur des permutations de grande taille, nous allons donc

utiliser une autre méthode : tirer au hasard des permutations. Pour permuter aléatoirement un tableau `t`, on applique juste la boucle suivante :

pour `i` de `0` à `n-1`
    tirer un nombre aléatoire `j` entre `0` et `i` (inclus) et échanger `t[i]` et `t[j]`

1. Écrire une méthode `permutationAleatoire(Tableau t)` qui permute aléatoirement un tableau `t`. On utilisera la fonction `Math.random()` qui tire un nombre aléatoire entre `0` (inclus) et `1` (exclus).
2. Écrire une méthode `complexiteMoyenneAlea(int n, int m)` qui calcule le nombre moyen de comparaisons et d'échanges effectués par nos deux algorithmes de tri lorsqu'ils trient `m` permutations aléatoires de taille `n`. Essayer avec `n` $= 5000$ et `m` $= 20$.
3. Quel semble être l'algorithme de tri le plus efficace entre le tri bulle et le tri rapide ?