TP n°4

Algorithmes de Tri et Complexité

L'objet de ce TP est de manipuler différents algorithmes de tris et de comparer leur efficacité. On se donne à chaque fois un tableau d'entiers et le problème consiste à modifier l'ordre des éléments de telle sorte qu'ils soient triés après l'application de l'algorithme. ***Pour chaque algorithme, la méthode correspondante doit a cher le nombre de comparaisons e ectuées ainsi que le temps d'exécution.***

Exercice 1 *Complexité*

1. Écrire une méthode `isSorted(int [] T)` qui indique si le tableau `T` donné en paramètre est trié par ordre croissant ou pas. Déterminer la complexité de votre algorithme dans le meilleur des cas et dans le pire des cas.

2. Considérer la méthode suivante :

```
public static int method1(int n){
        int a=n;
        int b=0;
        while (a>0) {
                if (a%2 == 1) then a--;
                else {
                        a = a/2;
                        b++;
                }
        }
        return b;
}
```

   Que fait cette méthode ? Déterminer sa complexité.

Exercice 2 *Tri par Sélection*

Le premier des algorithmes de tri est l'un des plus simples. Le principe est de séléctionner l'élément le plus petit du tableau , c'est-à-dire de trouver l'entier `p` tel que $\forall$ `i`, `T[i]` $\geq$ `T[p]`. Une fois cet emplacement trouvé, on échange les éléments `T[1]` et `T[p]`. Puis on recommence ces opérations sur le reste du tableau (pour les éléments compris entre les indices `2` et `n`. On recherche alors le plus petit élément de cette nouvelle suite de nombre et on l'échange avec `T[2]`. Et ainsi de suite . . .jusqu'au moment où l'on a placé tous les éléments du tableau.

Voici les états succéssifs du tableau `[9,5,1,6,2]` par cette méthode :

9	5	**1**	6	2
1	**5**	9	6	**2**
1	2	**9**	6	**5**
1	2	5	**6**	**9**
1	2	5	6	9

1. Quel est le pire des cas en nombre d'échanges pour la méthode précédente ?
2. À combien d'échanges procède-t-on alors ?
3. Quel est le nombre de comparaisons effectuées lors du tri dans le pire des cas ? Déterminer la complexité ?
4. Écrivez les méthodes suivantes :
   - une méthode `echange(int [] t, int p1 , int p2)` qui échange les éléments `t[p1]` et `t[p2]` du tableau.
   - une méthode `triSel(int [] t)` itérative qui réalise le tri par sélection.

Exercice 3 *Tri par Insertion*

Le tri par insertion est un algorithme que l'on peut qualifier de naif. Cet algorithme consiste à piocher une à une les valeurs du tableau et à les insérer, au bon endroit, dans le sous-tableau trié constitué des valeurs précédement piochées et triées. Les valeurs sont piochées dans l'ordre où elles apparaissent dans le tableau . Soit i l'indice de la valeur piochée , les $(i-1)$ premières valeurs du tableau constituent le sous-tableau trié dans lequel va être inséré la *i-eme* valeur. Au début de l'algorithme , il faut considérer que le sous-tableau constitué du seul premier élément est trié : c'est vrai puisque ce tableau ne comporte qu'un seul élément. Ensuite , on insére le second élément ($i = 2$), puis le troisième ($i = 3$) . . .etc. Ainsi, i varie de 2 à n, où n est le nombre total d'éléments du tableau.

Le problème de cet algorithme est qu'il faut parcourir le tableau trié pour savoir à quel endroit insérer le nouvel élément , puis décaler d'une case toutes les valeurs supérieures à l'éléments à insérer. En pratique, le sous-tableau classé est parcouru de droite à gauche , c'est à dire dans l'ordre décroissant. Les éléments sont donc décalés vers la droite tant que l'élément à insérer est plus petit qu'eux. Dés que l'élément à insérer est plus grand qu'un des éléments du tableau triée il n'y a plus de décalage, et l'élément est inséré dans la case laissée vacante par les éléments qui ont été décalés.

1. Appliquer le tri par insertion sur le tableau T=
5	3	1	6	4	2
.
2. Écrire une méthode `tri_inseration(int [] T)` qui utilise cet algorithme pour trier un tableau passé en paramètre.
3. Calculer le nombre de comparaisons dans le pire cas ?
4. En déduire la complexité ?

Exercice 4 *Jeu de la vie*

Le jeu de la vie a été inventé par John Conway dans les années 70. Ce *jeu* simule l'évolution de cellules dans un environement composé de cases. Il s'agit de donner une configuration initiale et de laisser évoluer le système.

L'environnement est un environement discrétisé à deux dimensions (tableau à deux dimensions). Chaque case du tableau peut être vide ou occupée par une cellule (et une seule !). Les cellules évoluent en fonction des ses voisins qui l'entourent (chaque cellules peut avoir 8 voisins directs). Les règles qui regissent l'évolution des cellules entre deux étapes sont les suivantes :

- **Règle de la survie :** Chaque cellule, ayant à l'étape `n` , deux ou trois cellules adjacentes survit à l'étape `n+1` ;
- **Règle de la mort :** Chaque cellule ayant pour voisines , à l'étape n, quatre cellules ou plus meurt par etouffement (surpopulation) à l'étape `n+1`. De même, une cellule qui n'a qu'une ou aucune cellule adjacente meurt par isolement.
- **Règle des naissances :** Chaque case vide , ayant exactement trois cellules adjacentes à l'étape `n` engendre une nouvelle cellule à l'étape `n+1`.

L'état à l'étape `n+1` d'une cellule est calculé uniquement à partir de l'état de la grille à l'étape `n` et non en fonction de la grille l'état de la grille à l'étape `n` augmenté des états à l'étape `n+1` déjà calculés.

Dans la version originale défini théoriquement par Conway, la grille est un plan infini, mais il est plus facile d'utiliser un tore pour simulation (la grille est de taille fixe , la ligne du haut est voisine de la ligne du bas , la colonne de gauche est voisine de la colonne de droite).

1. Définissez une classe `Population` modélisant une grille des cellules. Cette classe aura comme attributs : un tableau à deux dimensions de booléens et deux entiers représentant respectivement la largeur et l'hauteur de la grille. Tous les champs de cette classe auront un type d'accès `private`.
2. Écrivez un constructeur prenant la hauteur et la largeur de la grille et initialisant cette dernière sans aucun habitant. grille vide.
3. Écrivez une méthode `affichage()` qui affiche l'état de la grille sous forme textuelle (une chaîne de caractères par ligne, une étoile `"*"` pour une cellule vivante , un moins `" - "` pour une cellule morte).
4. Écrivez une méthode `setCellule(int i, int j)` permettant d'activer la case correspondante .
5. Écrivez une méthode `saisie()` qui initialise la grille par saisie des cases actives . La saisie se fera à l'aide d'une boucle `while` en demandant à chaque fois à l'utilisateur s'il veut continuer la saisie ; si oui on lui demandera les coordonnées de la cellule à activer.
6. Écrivez une méthode `nombreVoisins(int i,int j)` qui retourne le nombre de voisins vivants de la case située ligne `i` , colonne `j`.
7. Écrivez une méthode `generationSuivante()` qui calcule l'état de la grille à l'itération suivante.
8. Définissez une classe `JeudelaVie` qui contient la méthode `main()` et qui saisit une population de taille 10 par 10, l'affiche pour vérification, calcule (tant que l'utilisateur le désire) des générations en les affichant au fur et à mesure.