ANALYSES ASYMPTOTIQUES DES EXERCICES 1 ET 2 DU LABO 2

Exercice 1 : Problème des Tours d'Hanoï avec Récursion

```
/*
Algorithme récursif résoudant le problème des Tours d'Hanoï

   |          |          |
   |          |          |
   |          |          |
-------    -------    -------
   A          B          C

*/

public static void main(String[] args) {
    // Nombre de disques
    int N = 3;
    // A, B et C sont les noms des tours

    // Nous voulons d'abord résoudre le puzzle avant de le coder.
    // Solution Tours d'Hanoï avec 3 Disques:
    // 1. D1 de A vers C
    // 2. D2 de A vers B
    // 3. D1 de C vers B
    // 4. D3 de A vers C
    // 5. D1 de B vers A
    // 6. D2 de B vers C
    // 7. D1 de A vers C
    // Il y aura donc 7 print.

    // Pour commencer à coder notre solution, via un algorithme récursif,
    // nous allons évidemment structurer notre code selon le principe
    // fondamental de la récursion: la diminution de notre input (ici 'N').
    // Notre plus gros input ici est 3 (le disque 3), nous allons donc
```

```java
    // écrire notre code pour que nous implémentions le prochain mouvement (au départ
    // le premier mouvement) du disque 3 en premier récursivement. Il sera la base de
    // notre pile grandissante, laquelle comportera également le prochain
    // (au départ le premier) mouvement du disque 2 au milieu de la pile et
    // finalement le prochain mouvement du disque 1, ce dernier étant au sommet de la pile
    // il sera donc notre premier mouvement de la partie.

    // disque avec lequel on effectue le déplacement ---> paramètre 1
    // emplacement présent du disque -------------------> paramètre 2
    // destination du disque ---------------------------> paramètre 3
    // tour non-utilisée pour le déplacement -----------> paramètre 4

    // Notre premier appel à toursDeHanoi sera donc:
    toursDeHanoi(N, 'A', 'C', 'B');
}

public static void toursDeHanoi(int n,
                                char tourEmplacementInitial,
                                char tourNouvelEmplacement,
                                char tourAuxiliaire) {
    // Condition d'arrêt de la récursion (il n'y a pas de disque 0)
    if(n == 0) {
        return;
    }
    // On empile jusqu'à ce que nous soyons à notre condition d'arrêt.
    toursDeHanoi(n-1, tourEmplacementInitial, tourAuxiliaire, tourNouvelEmplacement);
    // Exemple: État du premier call stack (avant les mouvements simulés par le print):
    //          Pile_Base = (3, A, C, B)
    //          Pile 2 =    (2, A, B, C)
    //          Pile_Top =  (1, A, C, B)
    System.out.println(" Disque #"+n+" déplacé de la tour "+tourEmplacementInitial+
                       " vers la tour "+tourNouvelEmplacement);
    // Nouvelle appel à toursDeHanoi pour préparer le prochain mouvement du disque N-1
```

```java
     // (où N représente le disque venant d'être déplacé)
     // OU
     // pour dynamiquement ajuster le call stack si N était le disque 1
     // (c'est le cas au tout début de la partie lorsque l'on déplace le disque 1 au
     // sommet de notre pile originale)
     toursDeHanoi(n-1, tourAuxiliaire, tourNouvelEmplacement, tourEmplacementInitial);
 }

 // Analyse du BigO = À chaque appel récursif nous avons 2 possibilités:
 //                   L'algorithme se rappelle lui-même ou le thread actuel se
 //                   termine via la condition d'arrêt.
 //                   donc: (2 x 2 x 2 x ...) = 2^N
```

Exercice 2 : Tri Merge-Sort avec Récursion

```java
public static void main(String args[])
{
    // Le Merge Sort

    // Utilise le principe de diviser pour régner en divisant
    // constamment le tableau original en 2 sous-tableaux
    // pour pouvoir les recombiner triés à l'aide la récursion.

    int[] array = {8, 2, 5, 3, 4, 7, 6, 1};

    System.out.print("Tableau avant le tri: ");

    for(int i = 0; i < array.length; i++){
        System.out.print(array[i]+ " ");
    }

    System.out.println();

    mergeSort(array);

    System.out.print("Tableau après le tri: ");

    for(int i = 0; i < array.length; i++){
        System.out.print(array[i]+ " ");
    }
}
```

```java
private static void mergeSort(int[] array) {

    int length = array.length;

    // condition d'arrêt
    if (length <= 1) {
        return;
    }

    int middle = length / 2;
    int[] leftArray = new int[middle];
    int[] rightArray = new int[length - middle];

    int i = 0; // index du tableau gauche
    int j = 0; // index du tableau droite

    for(; i < length; i++) {
        if(i < middle) {
            leftArray[i] = array[i];
        }
        else {
            rightArray[j] = array[i];
            j++;
        }
    }
    mergeSort(leftArray);
    mergeSort(rightArray);
    merge(leftArray, rightArray, array);
}
```

```java
private static void merge(int[] leftArray, int[] rightArray, int[] array) {

    int leftSize = array.length / 2;
    int rightSize = array.length - leftSize;
    // Nos indices
    int i = 0, l = 0, r = 0;

    // On verifie les conditions pour le merge
    while(l < leftSize && r < rightSize) {
        if(leftArray[l] < rightArray[r]) {
            array[i] = leftArray[l];
            i++;
            l++;
        }
        else {
            array[i] = rightArray[r];
            i++;
            r++;
        }
    }
    while(l < leftSize) {
        array[i] = leftArray[l];
        i++;
        l++;
    }
    while(r < rightSize) {
        array[i] = rightArray[r];
        i++;
        r++;
    }
}
```

```java
    // Analyse du BigO = La complexité est ici en lien direct avec la taille de l'input
    //                   original. Au final chacun des éléments de l'input sera appelé
    //                   par la fonction linéaire aidante merge() via la division en 2
    //                   constante de l'input original, le résultat étant un arbre.
    //                   La méthode diviser pour régner étant appliqué pour chacun
    //                   des éléments nous aurons une complexité donc de O(n log(n)).

    // O(n log(n)) = merge-sort, quick-sort, heap-sort
    // O(n^2) = selection-sort, bubble-sort, insertion-sort

}
```