RAPPORT TP 1
COMPLEXITE

Réalisé Par :

- ***RIAZI Ibrahim - Groupe 03,***
- ***ZIDANI Fahed Imed - Groupe 01,***
- ***FELLOUSSI Idriss - Groupe 03,***
- ***RANGHEARD Léo - Groupe 03.***

Mini-Projet 1 : Calcul des nombres de Fibonacci.

On a implémenté différentes méthodes pour obtenir le nombre de Fibonacci.

- Version de base récursive.
- Version de base itérative.
- Version basée sur l'exponentiation de matrice.

Version de base récursive :

NOTE :
Dans notre Implémentation on a utilisé la classe BigInteger qui est utilisée pour les opérations mathématiques qui implique de très grands calculs d'entiers qui sont en dehors de la limite de tous les types de données primitifs disponibles. De cette façon, la classe BigInteger est très pratique à utiliser en raison de sa grande bibliothèque de méthodes et elle est également très utilisée dans la programmation compétitive.

Une méthode simple qui est une relation de récurrence mathématique à implémentation récursive directe .

```
public static BigInteger fiboRecursive(BigInteger n)
```

⇨ Complexité temporelle : exponentielle, car chaque fonction appelle deux autres fonctions.

Version de base itérative :

```
static BigInteger fiboIterative(BigInteger n) {
```

Tout d'abord, nous allons stocker 0 et 1 dans F[0] et F[1], respectivement.

Ensuite, nous allons parcourir les positions de tableau 2 à n-1. A chaque position i, on stocke la somme des deux valeurs du tableau précédent dans F[i].

Enfin, nous renvoyons la valeur de F[n-1], nous donnant le nombre à la position n dans la séquence.
L'analyse de la complexité temporelle de notre algorithme itératif est beaucoup plus simple que son homologue récursif.

Dans ce cas, notre opération la plus coûteuse est la cession. Premièrement, nos affectations de F[0] et F[1] coûtent O(1) chacune. Deuxièmement, notre boucle effectue une affectation par itération et s'exécute (n-1)-2 fois, ce qui coûte au total O(n-3) = O(n).

- ⇨ Par conséquent, notre algorithme itératif a une complexité temporelle de O(n).

C'est une nette amélioration par rapport à notre algorithme récursif

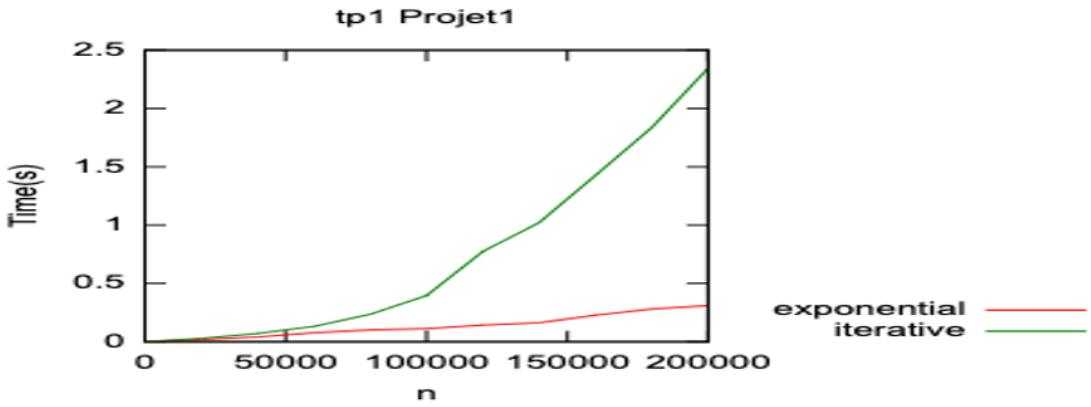
Version basée sur l'exponentiation de matrice :

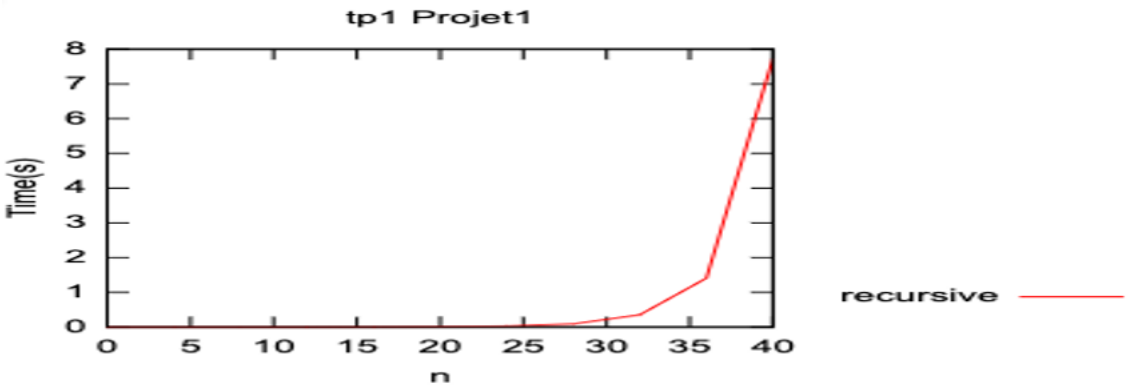
Cette version repose sur le fait que si nous multiplions n fois la matrice

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

par elle-même (en d'autres termes, calculons la puissance(M, n)), alors nous obtenons le (n+1)ème nombre de Fibonacci comme élément à la ligne et à la colonne (0, 0) dans la matrice résultante.en calculant la puissance en utilisant la méthode récursive.

- ⇨ On aura une complexité de O(Logn).





Conclusion :

- ⇨ la version de l'exponentiation de matrice et itérative vont plus vite que la version recursive
- ⇨ la version de l'exponentation de matrice va plus vite que l'itérative.

Mini-Projet 2 : Manipulation de graphes non-orientés.

On a implémenté les algorithmes suivants :

- Algorithme de vérification.
- Algorithme de Calcul de zone vide maximale.
- Algorithme de Calcul de zone vide maximum (méthode « complète »).
- Algorithme de Calcul de zone vide maximum (méthode « incomplète »).

NOTE :

- Dans notre Implémentation on a essayé d'éviter les méthodes prédéfinies de JAVA telle que [Les méthodes : de trie des listes (Sort), de vérification de l'existence (Contains, IndexOf, … etc.)], et cela pour bien préciser la complexité de nos algorithmes implémentés.
- Les tests d'efficacité sont dans la Class Main.

Test de vérification :

✓ On a conçu un algorithme de complexité O (n^2).

Calcul de zone vide maximale :

✓ On a conçu un algorithme de complexité O ($n*n^2$) = O(n^3).

✓ **n** pour le parcours mais à chaque parcours on lance **isZoneVide** qui est lui même **n^2** et par imbrication **O(n^3) .**

Calcul de zone vide maximum (méthode « complète »):

✓ On a conçu un algorithme de complexité O (n^2*n^2) = O(n^4).

✓ **n^2** pour le parcours mais à chaque parcours on lance **isZoneVide** qui est lui même **n^2** et par imbrication **O(n^4) .**

Calcul de zone vide maximum (méthode « incomplète ») :

➢ On a implémenté l'algorithme de **« Tri par Insertion »** afin de trier les sommets du graphe selon leur degré.

- dans le meilleur des cas le tri effectué au début de fonction **getSommetsTriés** soit en O(n), et dans le pire et le cas le tri effectué soit en $O(n^2)$.

✓ Donc la complexité de notre algorithme est : $O(n^2 + (n*n^2)) = O(n^3)$:

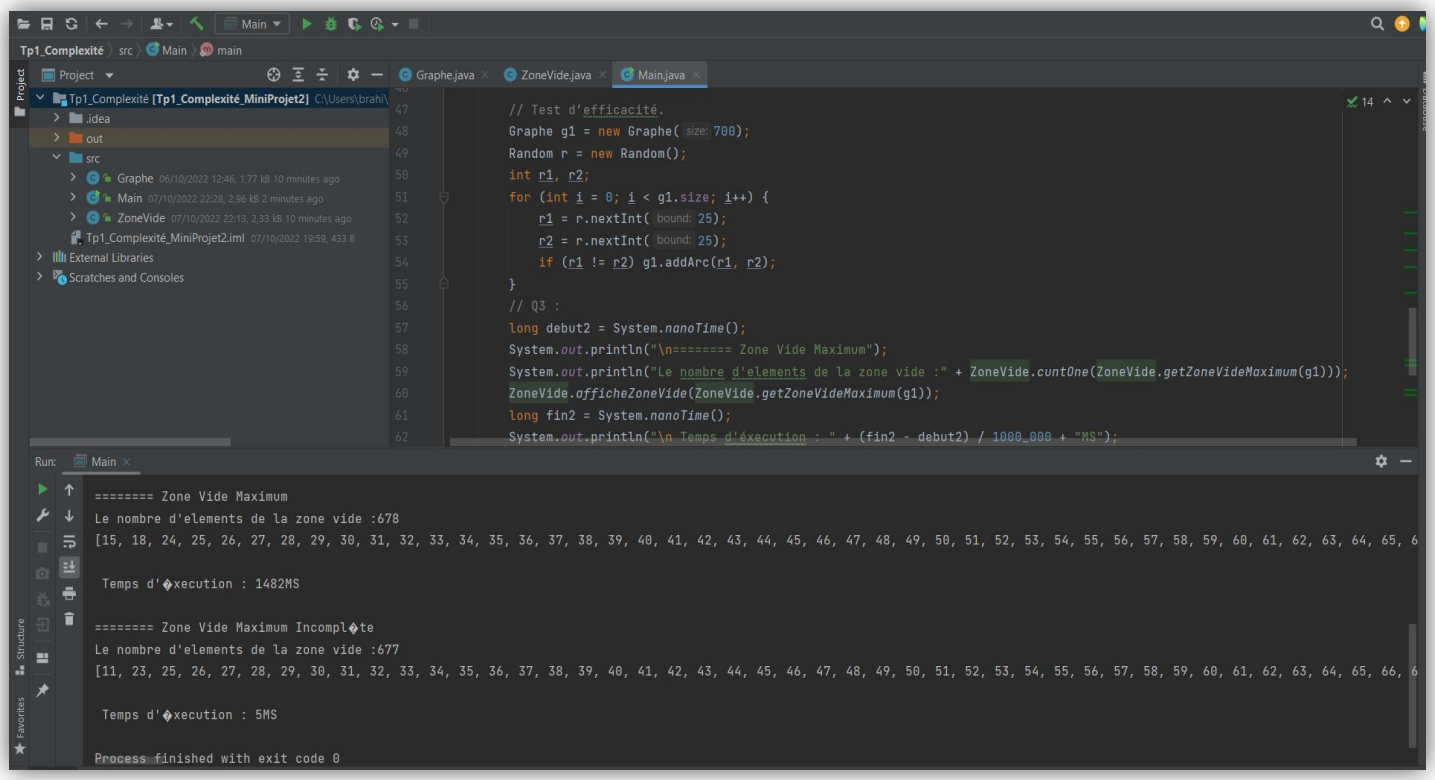
✓ **n^2** pour le tri et **n^2** pour le parcours des sommets triés et à chaque parcours on lance **getNeighbours** qui est lui même **n** et par imbrication **$O(n^2+n^3) = O(n^3)$.**

Tests d'efficacité :

On a testé l'efficacité de nos algorithmes en termes de matière d'exécution sur un graphe généré aléatoirement de 700 sommets **(Voir la Class Main)**.

On constate que le nombre d'éléments de la zone vide calculé par **getZoneVideMiximum** (678 éléments), est plus grand que le nombre calculé par **getZoneVideMiximumIncomplete** (677 éléments), par contre le temps d'exécution de **getZoneVideMiximumIncomplete** est plus petit que de **getZoneVideMiximum.**

getZoneVideMiximum ➔ 1482 Milliseconde .
getZoneVideMiximumIncomplete ➔ 5 Milliseconde .



Mini-Projet 2 : Simulation d'une Machine de Turing Déterministe**.**

En cette étape de mini projet, nous avons essayé de faire un simple simulateur de la machine de Turing qui à partir d'un mot et un programme (successions d'instruction), décide si l'entrée est acceptée ou pas . Le code était très simple : une classe qui dénote la notion de transition avec ses attributs et une classe principale Programme qui essaye de simuler le fonctionnement de la machine à travers ses méthodes principales.

- ✓ **lire ()** : une méthode qui importe les données depuis un fichier texte et remplit un **ArrayList** avec les transitions qui existe sur le fichier ainsi que les paramètres nécessaires pour la le fonctionnement i.e **(alphabet , mot à évaluer)** complexité : considérons toutes les opérations élémentaires(comparaisons , ajout dans le Arraylist, …etc.) de temps constant k , la fonction lire aura donc une complexité **linéaire** suivant le nombre de lignes de notre **fichier.txt** et donc **O(n)** .
- ✓ **réponse()** : une méthode qui parcourt simplement le mot en appliquant les règles de transitions sa complexité est de $\mathbf{O(n^2)}$ vu qu'on parcourt le tableau de transition de la manière faite ainsi on admet que les opérations élémentaires sont de complexité constante .
- ✓ Le **fichier txt** source du programme est conçu de manière de donner en première ligne l'alphabet utilisé puis en deuxième le mot et puis la successions des états transitions de façon que le premier champ est l'état puis suivit d'une transition