Extension : TAB

Spécifications

Tableaux

- Déclarer un Tableau : *Must Have*

Voici les trois manières de déclarer un tableau :

```
int[] array;
array = new int[taille];
```

```
int[] array = new int[taille];
```

```
int[] array = {0 , 1 , 2};
```

- Accès à un élément du tableau : *Must Have*

L'accès à un élément du tableau se fait un appelant son indice entre crochet, c'est-à-dire, comme ceci :

```
int element = array[indice];
```

- Méthodes natives sur les tableaux : *Must Have*

Il sera possible d'accéder à la longueur du tableau de cette manière :

```
int longueur = array.length;
```

Matrices

- Déclarer une matrice : *Must Have*

Voici les trois manières de déclarer une matrice :

```
int[] matrix;
matrix = new int[longueur, largeur];
```

```
int[] matrix = new int[longueur, largueur];
```

```
int[] matrix = {{0 , 1 , 2}, {3, 4, 5}};
```

- Accès à un élément de la matrice : *Must Have*

L'accès à un élément du tableau se fait un appelant son indice entre crochet, ligne puis colonne, c'est-à-dire, comme ceci :

```
int element = matrix[ligne, colonne];
```

- Méthodes natives sur les matrices : *Must Have*

Il sera possible d'accéder à la taille de la matrice de cette manière :

```
int longueur = matrix.length;
int largeur = matrix.width;
int taille = matrix.length;
```

Calcul Matriciel

- Matrice identité : *Nice to Have*

Il sera possible de créer une matrice identité de cette manière :

```
int[] matrix;
matrix = new int[longueur, largeur];
matrix.setIdentity();
```

```
int[] matrix = new int[longueur, largeur];
matrix.setIdentity();
```

- Matrice nulle : *Nice to Have*

Il sera possible de créer une matrice identité de cette manière :

```
int[] matrix;
matrix = new int[longueur, largeur];
matrix.setZero();
```

```
int[] matrix = new int[longueur, largeur];
matrix.setZero();
```

- Somme de matrices : *Must Have*

Il sera possible de faire la somme terme à terme de matrices de cette façon :

```
int[] matrixResult = matrix1.sum(matrix2); // À voir si on implémente M = M1 + M2
```

- Multiplication par une constante : *Must Have*

Il sera possible de multiplier une matrice par une constante de cette manière :

```
int constant = 2;
int[] matrixResult = matrix.multConst(constant); // À voir si on implémente M = constant * M1
```

- Produit matriciel : *Must Have*

Il sera possible d'effectuer un produit matriciel comme ceci :

```
int[] matrixResult = matrix1.matrixProd(matrix2); // À voir si on implémente M = M1 * M2
```

L'algorithme utilisé, afin d'avoir les meilleures performances possible, sera celui de Strassen.

- Transposée : *Should Have*

Il sera possible de calculer la transposée d'une matrice de cette façon :

```
int[] matrixResult = matrix.transpose();
```

- Déterminant : *Nice to Have*

Il sera possible de calculer un déterminant comme ceci :

```
float determinant = matrix.getDeterminant();
```

Pour calculer le déterminant rapidement, nous allons utilisé la décomposition LU, un des algorithmes les plus efficaces pour faire cela. De plus, on pourra envisager de stocker les matrices L et U afin d'optimiser l'inversion.

- Inverse : *Nice to Have*

Il sera possible de calculer une matrice inverse comme ceci :

```
float[] inverseMatrix = matrix.invert();
```

Tout comme le déterminant, nous calculerons l'inverse à partir de la décomposition LU de la matrice.

- Valeurs propres : *Nice to Have*

Il sera possible de calculer les valeurs propres d'une matrice comme ceci :

```
float[] eigenvalues = matrix.getEigenvalues();
```

Cette dernière spécification sera sans doute compliqué à mettre en place. Une piste pour les algorithmes à utilisés sont la méthode de Jacobi ou la méthode de Givens.