School of Computer Science and Informatics
Coursework Assessment Proforma

Module & Lecturer: CM2303: Algorithms and Data Structures, Bailin Deng

Date Set: 2nd February 2018

Submission Date: 2nd March 2018

Submission Arrangements:

Your coursework program – a Java application program in a file named `SparseMatrix.java` – should be submitted by logging into the coursework testbed at: `https://egeria.cs.cf.ac.uk/` before 9:30am on the submission date.

Make sure to include (as comments) your student number and your name at the top of your program file (`SparseMatrix.java`). Also, follow this by any notes (as comments) regarding your submission. For instance, specify here if your program does not generate the proper output or does not do it in the correct manner.

Title: Implementation of a Sparse Matrix Data Structure

This coursework is worth 10% of the total marks available for this module. The penalty for late or non-submission is an award of zero marks. You are reminded of the need to comply with Cardiff University's Student Guide to Academic Integrity. Your work should be submitted using the official Coursework Submission Cover sheet.

Instructions

Full instructions for this course work are provided on the following pages. Please make sure you understand them before you start.

Criteria for Assessment

Assessment and marking of a submitted program will be done by automatically checking the output of the submitted program against the solution program on a large set of test conditions. To verify this, you should use the testbed at `http://www.cs.cf.ac.uk/testbed/` which will allow you to upload your program and sample data files, and will then run both your program and the solution program, and compare their results. The testbed will also indicate where and/or how much the results differ.
It is highly recommended that you make extensive use of the testbed to check and refine your program – you can upload your program as many times as you like. The last uploaded version will be automatically taken as the version to be marked.
Credit will be awarded according to the functionalities that you correctly implement (see instructions on the following page for details of each functionality), as well as the time complexity of your implementation:

- 15% – loading and constructing sparse matrix information and elements from a file (5% for the matrix information (number of rows, columns, and non-zeros), 10% for the elements)
- 20% – transposing a sparse matrix
- 20% – adding two sparse matrices
- 20% – multiplying a sparse matrix by a scalar
- 20% – multiplying a sparse matrix with a dense vector
- 5% – efficiency (i.e. excessive run time will be penalised)

Further Details

Feedback on your coursework will address the above criteria and will be returned in approximately three working weeks. This will be supplemented with oral feedback in lectures and tutorials. If you have any questions relating to your individual solutions talk to the lecturer or the tutor.

Implementing a Sparse Matrix Data Structure

Matrices are a useful way to represent linear operations, and find application in most scientific and engineering fields. For example, in digital image processing, we can blur an image by updating each pixel to a weighted sum of itself and its eight neighbouring pixels. If we represent an image using an n-dimensional vector $\mathbf{r}$ with each element representing a pixel, then this blurring operation can be written as a n by n matrix $\mathbf{M}$, with each row representing the weights for summing a pixel and its neighbours. The blurred image $\mathbf{r}'$ can be written as the multiplication of M and $\mathbf{r}$, i.e., $\mathbf{r}' = \mathbf{M}\mathbf{r}$. In many applications, the matrix is 'sparse', meaning that most of its elements are zero. For example, the following is a sparse matrix:

$$\begin{bmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{bmatrix}. \tag{1}$$

The image blurring matrix mentioned above is also sparse, because for each row the weighted sum only involves one pixel and its eight neighbours. As a result, only nine out of the n elements in a row are non-zero. Nowadays pictures taken using mobile phones typically have more than one million pixels (i.e., n is larger than one million). Therefore, the image blurring matrix is highly sparse, with more than 99.9999% of its elements being zeros. To use such a sparse matrix in a computer program, it would be a huge waste of resource to store all its elements. Instead, we should only store its non-zero elements, as well as their positions within the matrix.

In this coursework, you will implement such a sparse matrix data structure. **For simplicity, we assume all the elements are intergers.** As part of the handout materials, you are already given a source file `SparseMatrix.java` that provides a skeleton for the class `SparseMatrix`. The class stores non-zero elements using a two-dimensional array

```
private ArrayList< ArrayList<Entry> > entries;
```

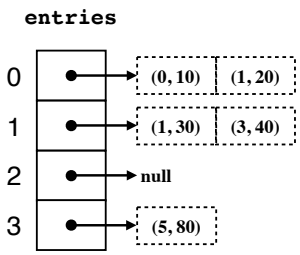
The size of the outer array is equal to the number of rows for the matrix. Each element of the outer array is of type `ArrayList<Entry>` and collects the non-zero elements of a row. The `Entry` class stores the (0-based) column index and the value of a non-zero element, and is already implemented in `SparseMatrix.java`:

```
private class Entry
{
    private int column;
    private int value;
    ...
}
```

For each instance of `ArrayList<Entry>`, the entries are stored in ascending order of their column index. When a row contains only zero elements, its corresponding `ArrayList<Entry>` instance will be either `null` or empty. Besides the `entries` array, the class also has a data member for the number of matrix columns

```
private int numCols;
```

Using this representation, the `entries` array for the sparse matrix in Equation (1) will conceptually look like the following figure.



In this coursework, you will complete the sparse matrix data structure by implementing the following functionalities.

1. **Loading matrices from files.** We will use a text file to store the information and elements of a sparse matrix. The first row of the file stores the number of rows and entries. After that, each row stores the row index, column index, and value for a non-zero element. For example, the text file for the sparse matrix in Equation (1) looks like

```
4 6
0 0 10
0 1 20
1 1 30
1 3 40
3 5 80
```

   The following method of `SparseMatrix` loads a matrix from a text file:

```
public void loadEntries(String filename).
```

   In `SparseMatrix.java`, there are already some codes inside this method to read content from the given file. You need to complete this method, by using the read content to set up the data member `entries`. **Remember that each `ArrayList<Entry>` for a row must store its entry in ascending order of their column index.**

   In addition, you need to complete the following method which returns the number of non-zero elements in the matrix:

```
public int numNonZeros().
```

2. **Transposing a matrix.** For a matrix M with m rows and n columns, its transpose M^T is a matrix with n rows and m columns, where the i-th row of M is the i-th column of M^T. For example, the transpose of the matrix in Equation (1) is

$$\begin{bmatrix} 10 & 0 & 0 & 0 \\ 20 & 30 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 40 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 80 \end{bmatrix}.$$

   You need to complete the following method which returns the transpose of the current sparse matrix:

```
public SparseMatrix transpose().
```

3. **Adding Two Matrices.** For two matrices A and B of the same size, their sum $C = A + B$ is a matrix where each element is the sum of the corresponding elements of A and B at the same position. For example,

$$\begin{bmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{bmatrix} + \begin{bmatrix} 0 & 30 & 0 & 25 & 0 & 0 \\ 0 & -3 & 0 & 0 & -5 & 0 \\ 0 & 0 & 2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 10 & 50 & 0 & 25 & 0 & 0 \\ 0 & 27 & 0 & 40 & -5 & 0 \\ 0 & 0 & 2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{bmatrix}.$$

   You need to complete the following method which implements this functionality:

```
public SparseMatrix add(SparseMatrix M).
```

   The returned value is a new `SparseMatrix` instance that is the sum of the current matrix and the argument `M`. You can assume the two matrices are of the same size. **You algorithm for constructing a row of the sum matrix must have time complexity no worse than $O(n_1 + n_2)$, where n_1 and n_2 are the**

number of non-zero elements in the corresponding rows from each matrix. If the time complexity of your implementation is worse than this limit, you will not get the 5% mark for efficiency. The returned matrix must be independent from both the current matrix and the matrix `M`, i.e., if either the current matrix or the matrix `M` is changed after this method is called, the returned matrix must not be affected.

4. **Matrix-Scalar Multiplication.** The multiplication of a matrix M and a scalar a results in a new matrix M' of the same size as M, where each element of M' equals to the corresponding element of M multiplied by a. For example,

$$\begin{bmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{bmatrix} \times 2 = \begin{bmatrix} 20 & 40 & 0 & 0 & 0 & 0 \\ 0 & 60 & 0 & 80 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 160 \end{bmatrix}$$

   You need to complete the following method which multiplies the current matrix with a scalar and updates `entries` according to the result.

```
public void multiplyBy(int scalar).
```

5. **Matrix-Vector Multiplication.** For an $m \times n$ matrix M and an n-dimensional vector v, their multiplication Mv is an m-dimensional vector f, such that its i-th element f_i is computed as

$$f_i = \sum_{j=0}^{n-1} M_{i,j} \times v_j,$$

   where $M_{i,j}$ is the element of matrix M at row i and column j, and v_j is the j-th element of v. For example,

$$\begin{bmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 20 \\ 150 \\ 0 \\ 400 \end{bmatrix}$$

   You need to complete the following method for this functionality

```
public DenseVector multiply(DenseVector v).
```

   Here `DenseVector` is a class that stores a vector and allows you to read or write its elements. It is already provided in `SparseMatrix.java`. The above method should return a vector that is the result of multiplication between the current matrix and the argument vector `v`. You can assume the size of the vector `v` is the same as the number of rows of the matrix. The returned vector must be independent of vector `t`, i.e., if `t` is changed after this method is called, the returned vector must not be affected.

Testing Your Program

The provided `SparseMatrix` class already implements a `main` method that allows you to run the application using different command line options. According to the options, it will run the methods mentioned above to perform different operations, and output the result.

- To load a matrix and print its information (number of rows, number of columns, number of non-zeros):

```
java SparseMatrix -i <MatrixFile>
```

  An example matrix file `MatrixInfo.txt` is provided as part of the coursework handout. You can also create your own files for testing.

- To load a matrix and print its elements:

  ```
  java SparseMatrix -r <MatrixFile>
  ```

  An example matrix file `MatrixRead.txt` is provided.

- To transpose a matrix:

  ```
  java SparseMatrix -t <MatrixFile>
  ```

  An example matrix file `MatrixTranspose.txt` is provided.

- To add two matrices:

  ```
  java SparseMatrix -a <MatrixFile1> <MatrixFile2>
  ```

  Two example matrix files `MatrixAdd1.txt` and `MatrixAdd2.txt` are provided. This command will also perform matrix-scalar multiplication to test your implementation.

- To perform matrix-vector multiplication:

  ```
  java SparseMatrix -v <MatrixFile> <VectorFile>
  ```

  An example matrix file `MatrixMultiply.txt` and an example vector file `VectorMultiply.txt` are provided.

You can use the testbed platform `http://www.cs.cf.ac.uk/testbed/` to upload your program and sample data files, and compare the output with the solution program. Both the `SparseMatrix` class and the `DenseVector` class provide a `print()` method that prints out their elements. You are not allowed to change these `print()` methods, as it will alter the output and result in different results compared to the solution output. Please also note that the provided example data files do not cover all the test cases that will be carried out when your submission is graded. Therefore, it is highly recommended that you test your program using more data.