The goal of this assignment is to become comfortable using the basic Array and ArrayList data structures and to gain some exposure to sparse matrices. A sparse matrix is a data structure which does not require any memory to represent a 0.0 entry, which is useful for low-density data. We’ll run a simple algorithm on the two matrix types as a simple quantitative experiment to compare the execution times of your two implementations.

1. [10] Open Eclipse and create a new project (File -> New -> Project -> Java -> Java Project) named “Assignment3” (keep all other default settings). When the program starts, prompt the user for an integer matrix dimension (we’ll be using square matrices), a double-precision matrix density, and an integer iteration count (we’ll run the algorithm on many different matrices to determine an average execution time). For example, if the user enters a dimension of 10, a density of 0.1, and an iteration count of 20, your code will create 20 dense and 20 sparse 10x10 matrices, each of which contains an average of 10% non-zero entries.

2. [80] For each of the above iterations (i.e., specified by the iteration count), do the following:

   a. (20) Create two data structures to represent matrices. The first structure should be a 2-dimensional array of type double to act as a dense matrix. This 2-dimensional array will be square and it’s dimensions based on the above matrix dimension (i.e., dimension x dimension). The second structure should be based on ArrayLists, and should also be of type double. This structure will be used to represent all of the non-zero entries in the dense matrix. You can organize this second structure any way you’d like, provided that it contains no 0.0 entries. Note that if you’re not doing the extra credit, you won’t need to store any information related to the entries other than their values. This is because you’ll simply be calculating row sums (see part d below). To do the extra credit (i.e., matrix multiplication), you’ll need to store the row and column of each entry. I recommend writing a class to store the row, column, and value for each entry.

   b. (10) Populate the dense matrix by looping through the data structure and first determining whether or not each cell should contain a nonzero entry. To do this, generate a random number on the range [0.0, 1.0) using Math.random() and then compare that number against the desired density (from part 1). If Math.random() <= density, then populate the cell with the result of Math.random() + Double.MIN_VALUE (the epsilon prevents 0.0 values); if not, populate the cell with 0.0.

   c. (10) Populate the sparse matrix such that it contains the same values at the dense matrix above expect that there are no entries in the data structure for the 0.0 entries. You may either accomplish this while populating the dense matrix or by first generating the dense matrix and
then copying the values into the sparse matrix. Keep track of the actual number of non-zero cells.

d. (20) Write an algorithm to determine which row in the dense matrix contains the largest row sum (i.e., the sum of the values in the row). Keep track of the largest row sum and execution time taken for this operation (see System.nanoTime()) so that you can later report the average largest row sum and total execution time. See the extra credit for a more interesting matrix operation.

e. (20) Write an algorithm to determine which row in the sparse matrix contains the largest row sum. Keep track of the largest row sum and execution time taken for this operation so that you can later report the average largest row sum and total execution time

3. [10] Output the total time taken for the dense and sparse operations (in nanoseconds) along with the average number of entries in the sparse matrices and the average largest row sums. You output should look something like (dimension of 1000, density of 0.01, 100 iterations):

average dense cells: 1000000
average sparse cells: 10000.17
average dense max row: 11.931248231666707
average sparse max row: 11.931248231666707
total dense time: 101202000
total sparse time: 15282000

Observe the differences in execution times as you vary the matrix dimension and number of iterations. List and analyze your observations in the comments of your code.

4. [+20] (extra credit) In addition to finding and printing the row sum of the matrices, also calculate the product of each matrix multiplied by itself (i.e., implement matrix multiplication on dense and sparse matrices).

Zip the Assignment3 folder in your Eclipse workspace directory and upload the .zip file to Blackboard (see Assignment 3 assignment in the Course Documents area).