Koon Chua
EN 605.202.81
7/17/2018
Lab 2 Analysis

I. Introduction:

In Lab 2, our objective was to compute the determinant of a $n\ x\ n$ matrix. This consists of a few key steps:

- Iterating through an input file, checking the validity of its components
- Creating our own Matrix ADT to be used to manipulate the input matrix
- Calculating the cofactor and determinant of each input matrix
- Writing the results in a cogent format for review

With these objectives laid out, I created four classes to tackle each individual sub-problem.

ReadWrite combined both the reading/writing processes into a single class. It uses Java's built-in File class to open the input and output files, Scanner class to parse through the input file, and PrintWriter class to write and close the output file. The output file first prints the matrix, then its determinant below.

The ArrayMatrix class contains an array implementation of a matrix. It contains multiple methods:

- Value: returns value at a given (i,j) indices
- insertVal: inserts value at a given (i,j) indices
- checkSquare return true if matrix is square
- numRow/numCol return number of rows/columns
- toString returns String representation of matrix

The Determinant class contains the recursive function for calculating the determinant for our input matrix. It also contains a function to compute the cofactor as a necessary step in computing the determinant.

The main driver class Lab1Main initializes instances of these classes, and also captures errors due to invalid input/output files or incorrect parameters.

II. Implementation Justification:

While we were required to use an array-implementation, I think this implementation makes the most logical sense given the nature of our problem. We needed to access different locations with the array, and random access simplified that process.

Because recursion was also a requirement, the computeDeterminant function utilizes a Laplace Expansion, where the determinant of a nxn matrix is equal to the sum of the scaled minors of the matrix. This is defined as: $\sum_{j=1}^{n}(-1)^{i+j}a_1M_{1_j}$, where M_{1_j} is the cofactor of the matrix N after removing row 1 and column j. We recursively reduce our matrix into its cofactors, and sum the determinants.

III. Efficiency:

The cost of our recursive determinant algorithm is O(n!). In order to get to our final answer, we have to repeatedly sum the cofactors of our matrix until we finally hit our base case of n = 1. (i.e. n * (n-1) * (n-2) … * 1). So while our algorithm is concise and efficient for smaller n, it quickly becomes computationally expensive as n increases. One way to reduce this computation time is by implementing a hybrid iterative algorithm. We might also be able to take advantage of memoization by using a cofactor "cache" so we can avoid recomputing the same cofactors repeatedly.

IV. Iterative Comparison:

As the recursive solution costs O(n!) to compute, it becomes expensive very quickly. We could try to implement an iterative solution, where the determinant is formed from the product of smaller determinants that are easier to compute (n = 1, 2, 3, etc.). With an iterative or hybrid approach, our computation time for large n is drastically improved since we don't need to compute as far down as our recursive base case.

V. Future Iterations:

In a future iteration I would like to try and code a solution using LU decomposition (Gaussian Elimination). With this method, the matrix A is broken up into a lower and upper triangular matrix (and a possible permutation matrix),

$$A = P^{-1}LU$$

We can then define the determinant of A as follows:

$$\det(A) = \det(P^{-1})\det(L)\det(U) = (-1)^s \prod_{i=1}^{n} l_{ii} \prod_{j=1}^{n} l_{jj}$$

Since, the determinant of a triangular matrix is just the product of its diagonal entries.

Compared to our recursive Laplace expansion solution, the matrix decomposition is performed in $O(n^3)$. Therefore, this solution would be more efficient when working with larger matrices.