Candace Holcombe-Volke

Lab 2 “Determinants” Reflection

In this analysis I will discuss the implementation of the program to find determinants of matrices of variable size, the recursive procedure used, error checking included, possible iterative alternatives, and lessons learned.

My program to find the determinants of a matrix collects command line arguments from the user, including the output file location, followed by any number of input files containing matrices to calculate. From the ReadFile class, the main() method is executed which collects the arguments and calls the readFile() method, passing each input file from within a loop of all input files. It is the responsibility of readFile to pull in the data from the input files one line at a time, and determine whether the input data is intended to be a notation of the degree of the subsequent matrix, or data within the matrix itself. This is complicated for matrices of one degree, and due to such a matrix being indistinguishable from the degree notation itself if only based on the length of entry, the program makes use of the fact that matrix elements are followed by a space character, where degree notations are not. The method assigns the degree value to the order variable when appropriate, and otherwise, pulls in data to a temporary one-dimensional array called tempMatrix. The String input is parsed into int elements and assigned to the main matrix, called readMatrix. As each subsequent line is read from the input file, the method checks that it contains the space character, increments the row index of the readMatrix, and then assigns to the main matrix in the next row. Once the number of rows of readMatrix reaches the appropriate size based on the order read from the input file, the function passes readMatrix and the order to getDeterminant().

GetDeterminant(), in combination with getMatrix() uses a recursive algorithm to calculate the determinant of each matrix. First, getDeterminant checks to see if the order of the matrix is equal to 1, in which case it will return the only element of the function as the determinant. This is the function’s only stopping case. If the order of the value is greater than 1, then the recursion begins. getDeterminant accepts two arguments, the starting matrix, which is a two-dimensional array, and an int value of the order of the matrix. The argument bypasses the stopping case, and then enters a for-loop which iterates across the matrix from left to right in the top row. For each column, the method will calculate the newDeterminant as the submatrix multiplied by the multiplier for all submatrices of one degree less than the original matrix. newDeterminant is summed into determinant, executing the Laplace formula. While the degree continues to be greater than 1, getDeterminant will continue to recursively call itself decrementing the degree by 1, and substituting the startingMatrix with the submatrix return of getMatrix. Once the method finally reaches a call where the degree is equal to 1, it will return the value of the one-degree matrix, and begin to reverse, increase the degree of the matrix, and pull in the values of each submatrix determinant, starting with all degree 1 submatrices.

As the determinants are calculated, the results are returned to readFile, who in turn, calls writeFile to print to an output file.

While getMatrix is not written as a recursive method, it could be done so with only minor changes. Currently, getMatrix iterates through each column along the top row of the original matrix passed through it to find the location of the elements required to build the submatrices. However, this could be made recursive by setting the stopping case to be when the column index is equal to the order-1, at which point it could build the submatrix, and call getMatrix again with the column index equal to the current column-1 while the column >=0.

Candace Holcombe-Volke
Lab 2 - Determinants

This program uses a variety of error checking to ensure that the proper results are produced. Firstly, it inspects the length of tempMatrix, which is the one-dimensional array that holds the data of one line of input split based on the space character, and compares it against the noted matrix degree. If the degree specified in the input file does not match the length of tempMatrix, it signals that the matrix does not have enough, or has too many, columns. In this case, it will return an error message and continue to the next degree input. Additionally, the program ensures that the matrix does not have more rows than the required amount for its noted degree. If readFile collects more rows of elements for a matrix than expected, it will catch the ArrayIndexOutOfBoundsException and return a specific message indicating that there are too many rows.

The program also has protections against non numeric element entries. If an element is not numeric, the program will catch the NumberFormatException, return an error message, and continue to the next matrix. Finally, the program will check that the degree stated by the input file is at least 1. If the degree is less than 1, it will return an error message because that is not a valid matrix.

While this program makes use of a recursive algorithm to calculate the determinants, there are iterative solutions that could be implemented as well, however I do believe that in this case they would likely be more complicated to program than the recursive design. An iterative solution would require multiple nested loops to move through each matrix. Assuming that the input matrix was of degree 2, we would have to cross the matrix from left to right, collecting each element in the top row as the multiplier to the single element *not* in its column. This would make use of row and column variables, and a for loop to cross left to right. Additionally, the determinants of each of the one-degree matrices would need to be summed.

Now, when the matrix grows beyond a two-degree item, the iteration becomes more complicated. The program now requires iteration from left to right and top to bottom. Nesting two for loops, the program would need to move from the left, collect the element in the top left corner as the first multiplier. Next, the method would increment both row and column by 1 moving down and to the right one element. It would need to store the value of this element in another multiplier variable before incrementing row and column again. If the program had reached a one-degree matrix at this point, it could return that value, which would be multiplied by the most recently collected multiplier. A stack would be a good storage container for this process because the most recent multiplier to have been added will be the next one to be needed, demonstrating the stacks FIFO structure. The next loop will now increment the column by 1, collecting the value of the multiplier in that column, and returning the value of the single element in the new one-degree matrix. The two determinants are summed (accounting for the alternation of positive and negative coefficients), and the program returns to the three-degree matrix. The loop continues by incrementing the column (currently == 0) by one, and continuing the process of finding the determinants of the two-degree submatrix and summing with the previous determinants.

Through the process of developing this program, I learned a lot about the way recursive programs operate. I found some parts quite challenging, such as combining loops with recursive elements, and found that I was tending to try to “control” too much of the program. One example was my attempt at controlling the sign of the coefficient. In the end, I implemented a very simple solution into the calculation of the getDeterminant recursive call. One way I think I could make the program more efficient is to eliminate the getMatrix method which recreates a smaller matrix for every recursive call. In order to do this, I would need to find a better way to manage the movement through the matrix,

by finding ways to refer very generally to locations among the columns and rows. Recreating the matrix allowed for a very convenient stopping case return of matrix[0][0], but that wouldn't be the case if each submatrix weren't its own entity, but rather a "marked off" area of the original matrix.

From a readability and clarity standpoint, the recursive algorithm is much more straightforward than the iterative, and I can certainly see the benefit of that. However, the recursive implementation I believe has a cost of O(n!), which is quite expensive, particularly if using anything other than very small matrices.