EN.605.202.81.SU18 Data Structures**: Introduction to Data Structures**

Aravind Ravindranath

Determinant of a Square Matrix using Linked List (LAB 3)

Due Date: July 31, 2018

Dated Turned In: July 29, 2018

Analysis of Matrix determination calculation

The current requirement deals with calculating the determinant of a square matrix. The program should work for a square matrix of order 6 and below. It was required to use a Linked List Approach for storing the definition of the matrix and doing operations on it.

The design I chose was to have a multi linked list with a header node. The header node points to the element in the first row and first column of the matrix. Each node which represents an element of the matrix has references to two other nodes, one to the right which is the next element in the row and one to the bottom element in the same column. The diagram below represents the idea. The subscript nm means it's the element in the m^{th} row and n^{th} column.

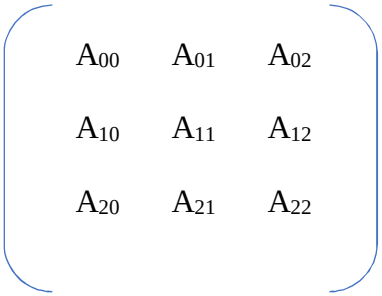

Figure 1 3rd order square matrix

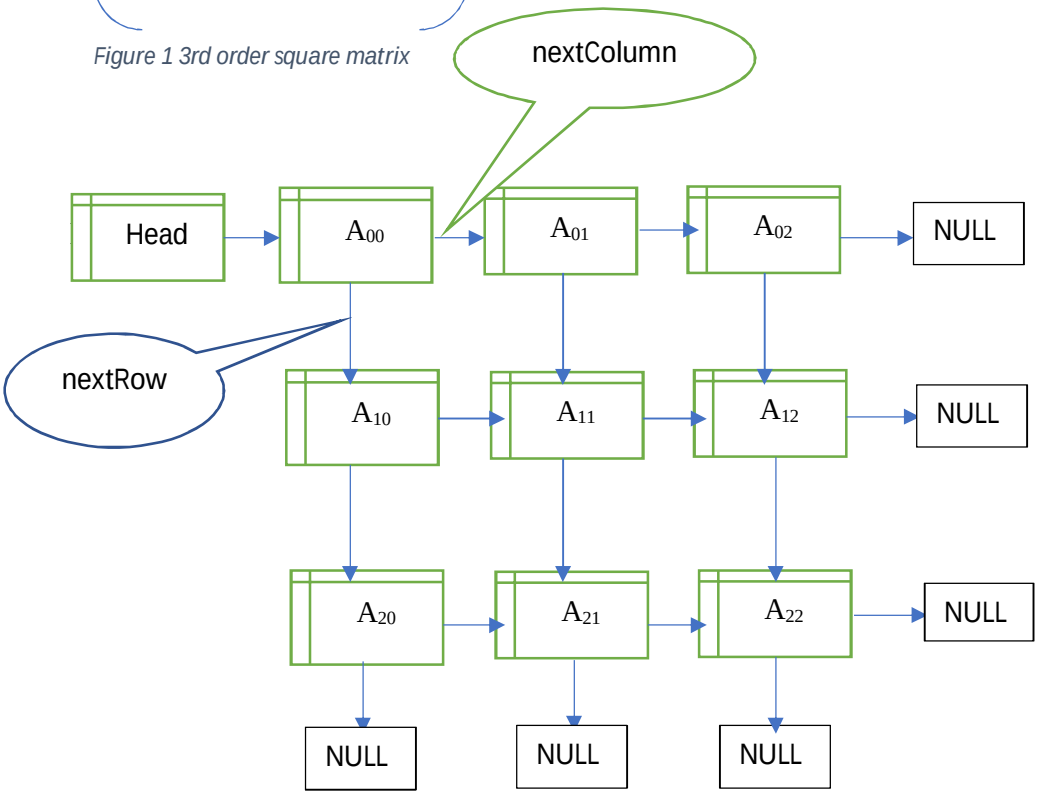

Figure 2 Multi-Linked List representation of the 3rd order square matrix

As can be seen in the diagram above, the Header represented by head holds the reference to the element in the first row and first column of the matrix.

Each element of the matrix, then has a reference to the element to its immediate right and to its bottom. This representation kind of depicts a forward and downward traversal which helps in determining the minors of the matrix. The head node provides the reference to the topmost and the leftmost element and from there traversal to any element becomes possible. Most of the times we need access to the beginning of a row and that is achieved by going from the head to the next row and the next row and so forth. Within a row as well, we move rightwards across elements by referencing the next column pointer. The determinant calculation is performed with the first row as the pivot and the traversal will be to the right and to the bottom to find the minor matrix.

When we compare this with the array representation, we lose the random-access capability. When dealing with matrix operations, random access comes quite handy. With multi linked lists, we must do traversals via node references to reach the right element. On an average, since there are n^2 elements in the matrix, we can expect an average amount of n/2 row traversals and n/2 column traversals leading to an $O(n/2 + n/2) \approx O(n)$ cost for retrieving one element. Now if we need to access all the n elements then there will be $O(n) * n \approx O(n^2)$ cost.
With multi linked list, I observed that lot of guard rails must be built to prevent Null Pointer Exception, where as in an array, knowing an order out of bounds exception can be prevented easily.

1 Brief description of the project and the objects:

I structured the entire project into four packages.

Package **com.jhu.ds.lab3.MatrixOperations**: This package consists of a class `MyLinkedMatrix` which contains the necessary attributes and methods to build and display square matrices. The two main methods help in creating minor matrices and calculating determinants. This class uses objects of class `ElementNode` to represent the elements of the matrix. The matrix is built using a Multi Linked List Implementation. The determinant is calculated using recursive formula based on Laplace expansion.

Package **com.jhu.ds. lab3.MatrixOperations.Iterative**: This package consists of a class `MyLinkedMatrix`, like the class in the above package. The difference is in calculating determinant where an iterative formula based on modified Gauss-Jordan elimination is used.

Package **com.jhu.ds.lab3.MatrixReuse:** This package consists of an abstract class `MyAbsMatrixDeterminant` which is extended by the class MyLinkedMatrix. The abstraction helps in invoking the recursive and the iterative calculation in the method of class ProcessInputSquareMatrices, on request without the need for additional code branches tapping into polymorphic behavior.

Package **com.jhu.ds.lab3.IOoperations**: The class ProcessInputSquareMatrices is the driver class which interacts with the end user and orchestrates the call to Matrix Operations class.

In addition, this project consists of JUNIT test cases which provides a guard rail for the functions and features developed in this project. The test cases test the main algorithms in

com.jhu.ds.lab3.MatrixOperations.Iterative.MyLinkedMatrix and com.jhu.ds.lab3.MatrixOperations.MyLinkedMatrix class. The tests reside in package com.jhu.ds.matrix.test .

2 Algorithms used:

2.1 Recursive Method

Recursion comes naturally to calculation of determinants if we use the formula mentioned which is as follows:

$$\mathbf{det}(A) = \sum_j (-1)^{i+j} * A[I,j] * \det(\text{minor}(A[I,j])), \text{ for any } j$$

The cost of calculating the determinant using the above recursion was already calculated in the previous lab with array representation and it approximated to about O(n!). This holds true even in the case of the linked list based matrix representation. But there is some additional cost which gets factored since we do a lot of traversals in finding the co-factors. While building the minor matrix, in my logic I do a two-level nesting with a few reference assignment operations. Compared to an array-based project, there is some additional overhead which kicks in. If there are n rows and n columns, on an average there could be n/2 row and n/2 column traversals and due to 2 level nesting it leads to $O(n^2/4) \approx O(n^2)$. Now building of the minor works alongside the calculation of the determinant in the recursive call and therefore, we can state that the overall cost will be $O(n^2) * O(n!) \approx O(n^2*n!)$.

Correction: After some careful observation, I have come to the conclusion that even for an array based matrix the cost comes closer to $O(n^2*n!)$. I did some comparisons as I have depicted below in the runtime measurements. My initial approximation of O(n!) even though not incorrect, did not account for the nested for loops getting executed in the recursive calls for the array based representation.

2.2 Iterative Approach

In the previous lab, I tried to iteratively build a logic to implement Laplacian expansion. The logic was too complex and cumbersome. At the end, the estimated cost came out to O(n!). I did some research primarily on the internet and came across row reduction methods which led to be Gauss-Jordan elimination technique to calculate the determinant. I decided to use a slightly modified Gaussian approach. To build a robust algorithm I decided not to do any row or column swaps. The primary target was to build an upper triangular square matrix, which then would lead to an easy calculation of determinants, as a product of the diagonal elements. Method `calcDeterminantGaussian` in class `com.jhu.ds.lab3.MatrixOperations.Iterative.MyLinkedMatrix` holds the logic. When I did runtime measurements of this algorithm I was surprised by the response time. I was able to calculate the determinant of a 100*100 square matrix quite fast. With the recursive method, it was taking several minutes for a 13*13 matrix.

While analyzing the cost, it was clear that the runtime is much lesser than O(n!). In the implementation I have nested loops, with a maximum of two levels of nesting. So, with n as the loop limit, there was roughly n*n*n operations performed due to the nested loops (loop within a loop within a loop). Thus, the approximate cost would be $O(n^3)$. After some tracing I found that

the actual runtime is slightly more than $O(n^3)$ and less than $O(n^4)$. With an approximation, I was able to fit the measurement points on to $O(n^{7/2})$. This makes some sense since there are a few nested loops and when they add up they tend to what is observed.

3 Learnings:

As in the previous lab using array representation, recursion leads to a highly non-linear behavior. The moment we start calculating the determinant of a 13*13 matrix, there is a massive slow down. Recursion is elegant from an implementation perspective but due to the prohibitive cost, at medium to large data sets the performance degrades. There is not much scope to improve due to the recursive nature. For comparison, when we implement an iterative solution with Gauss-Jordan elimination, the response times are significantly better. Even in the latter approach, we come across non-linear behavior, but it is a lower power of n, close to $O(n^3)$. $O(n^3)$ is much better than O(n!) and it is observed that even calculating a determinant of 100*100 square matrix, the response time is quite reasonable. In conclusion, an iterative implementation would be more feasible when we come across use cases where we tend to tackle problems with large number of variables and/or large data sets. If the use case deals with smaller data sets, recursion works just fine.

The Linked List representation of the matrix, introduces some amount of complexity during implementation. The references must be built correctly, and Null Pointer Exceptions should be avoided. To reach an element, node to node traversal is the only way out.

When dealing with square matrices, since the order is given, the need for dynamic memory allocation does not arise thus not providing any additional advantage for the use case of calculating determinants. For the specific use case of calculating determinants, array-based representation seems to offer an easier and more comprehensible code lines than that of Linked List based implementation.

4 Runtime Measurements of recursive and iterative algorithm:

A class utility has been developed to measure the runtime of the algorithms. The class `MeasureDeterminantCalcRun` for the recursive algorithm and `MeasureDeterminantCalcGaussianRun` for the iterative algorithm, does runtime measurements. It resides in com.jhu.ds.matrix.test package.

4.1 Recursive run

Size	Time(ns)
2	20527
3	26775
6	2617248
8	24677540
10	971295513
11	9967513557
12	1.05644E+11

Table 1 Runtime of the recursive method in nano seconds, size is order of the square matrix

As is explained with the algorithm analysis, the plotted points show an O(n!) cost trend. A graph of the factorial function is also plotted alongside to show the trend. When I plotted $n^2*n!$ graph, the trend came closer to the points plotted. Therefore, to be more accurate the cost approximates to $O(n^2*n!)$. The graphs are plotted by using the free utility from the web desmos.com/calculator

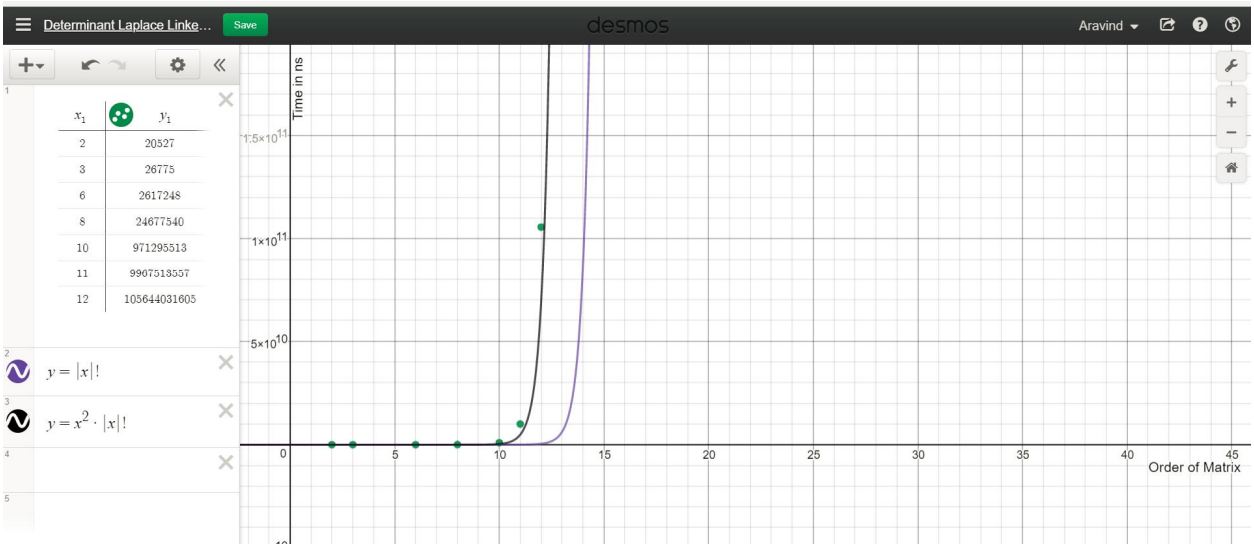

Figure 3 Runtime measurement of recursive algorithm

4.2 Iterative run

Size	Time(ns)
2	12941
4	16065
6	28113
8	57120
10	84341
16	316836
20	3200494
30	2092013
40	3036721
100	10059334

Table 2 Runtime of the iterative method in nano seconds, size is order of the square matrix

The iterative algorithm also displays a nonlinear runtime and the cost approximates to O(n3) and fits reasonably well to a plot of $n^{7/2}$. But as can be seen the response is much better than that of the recursive method.

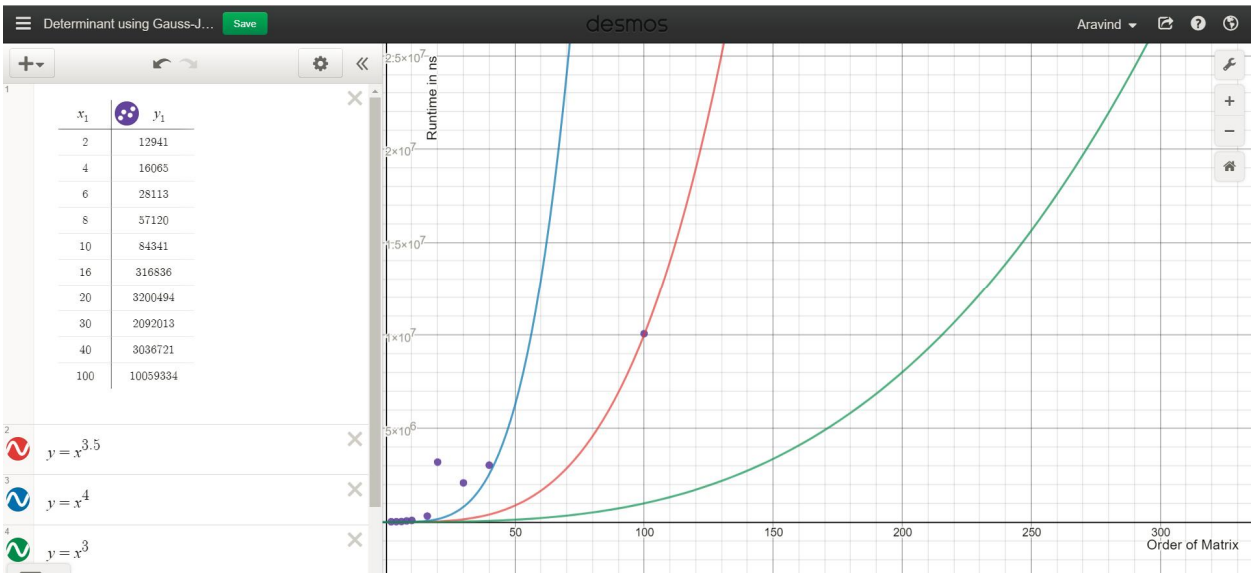

Figure 4 Runtime measurement of iterative algorithm