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

Aravind Ravindranath

Determinant of a Square Matrix using Arrays(LAB 2)

Due Date: July 17, 2018

Dated Turned In: July 15, 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 an array for storing the definition of the matrix and doing operations on it.

In my implementation I chose to go with a 2-dimensional array of integers. In matrix operations there will be a need to a lot of random access based on indices. Array based representation is ideal for such random access with a constant runtime. The disadvantage of having to allocate memory in advance for an array does not affect the use case of matrix operations in this project. The input is always a square matrix of known order.
We could use a linked stack or queue for this as well. But the amount of retrievals and the need to build submatrices leading to other stacks and queues would lead to some implementation challenges. The scope is to find determinants recursively. Since the stacks and queues are based on a FIFO/LIFO concept, random access would not have been possible other than going through a pop() operation.
We could of course use linked lists which allows representation of an array especially a multi linked list which provides navigation possibilities. We must sequentially go through the multi linked list to find the value at a row and column linked list but it is still more elegant than stack and queue based approach. The cost of reading the right element will be O(n) exhibiting a linear behavior, where n is the order. The advantage of being able to dynamically allocate memory will not be utilized here as there is no use case for increasing size of the matrix.

1. Brief description of the project and the objects:

I structured the entire project into two packages one package containing all the aspects for the matrix operations and the second one which had a class dealing with Input/Output tasks.

Package **com.jhu.ds.lab2.MatrixOperations**: This package consists of a class SquareMatrix 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

Package **com.jhu.ds.lab2.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 SquareMatrix class. The tests reside in package com.jhu.ds.matrix.test .

2. Algorithm used and comparison with iterative approach:

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

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

The determinant of a matrix leads to a calculation of determinants of its submatrices until it reaches a stage where the submatrix has order 1. This is the base case where the determinant is the element itself for a1st order matrix. The formula comes with a recursive pattern. The recursion is helped by another method which returns the minor of an element whose determinant needs to be calculated. Now for an n order matrix, we would have n operations, each operation needing determinant of a (n-1) submatrix. This (n-1) submatrix would have (n-1) operations leading to a determinant calculation of (n-2) submatrices and so on and so forth, until we reach the base case of 1*1 matrix.
Now we observe that for calculation of a determinant of an element of nth order matrix, it takes (n-1)! Operations = (n-1)*(n-2)*(n-3)*……*1 operations. Since we have to calculate n determinants to derive the determinant of the n*n matrix we will have n*(n-1)! operations which is nothing but n! operations. Therefore, the recursive approach in calculating determinants leads to O(n!) runtime. And in the final section taking some runtime measurements it exhibits this pattern and from determinant of 13*13 square matrix, the program was not responding.

Iterative Approach

I tried to come up with an iterative approach and realized that it is cumbersome to do so if the above formula needs to be applied. As the above approach is naturally recursive, to break it out into an iterative set of operations is not ideal and makes the logic less elegant.
Depending on the order of the matrix, we must iteratively calculate the determinant and store it in an appropriate data structure in memory and then perform the multiplication and the summation.

The logic would be pre-calculate the determinants starting from the minor matrices of order 1. After this is determined then we go on calculating the second order minor determinants, so on and so forth until we find determinants of submatrices of the (n-1)th order. After this it will be a summation of the signed product of the element and its minor's determinant.
A doubly linked list for holding determinants could be used which would fit well as there will be a need to go back and forward depending on the element for which the determinant is required. An array could be used as well based on the algorithmic approach

A class needs to be defined as mentioned below which will be the data part:

```
Class ElementDeterminantNode{
        int  row,  //row position of the element in the main n*n matrix
        int col,    // column position of the element in the main n*n matrix
        int suborder  // order of the minor matrix for which the determinant is calculated
        int determinant //determinant of the minor matrix
        ElementDeterminantNode next
        ElementDeterminantNode prev
}
```

A doubly linked list can be used as described below or an array holding the object of class described above or an array of type ElementDeterminantNode:

```
Class DList{
        Element Determinant head
        Element Determinant tail
        int size;
}
```

For the sake of simplicity, I am including a pseudo code which I have not programmed yet but reasonably confident that it shall work.

//taking the first row as the pivot for Laplacian approach. Assuming DetArray and Temp are arrays for the sake of simplicity.

```
ElementDeterminantNode[]  DetArray = new ElementDeterminantNode[n];
ElementDeterminantNode[]  Temp = new ElementDeterminantNode[n];

int i = 0;
while( i < n-1){
        int j = 0;
        if ( n > 0){
                clear DetArray;
                DetArray = Temp;
        }
        do n times{
                if DetArray is initial
                        ElementDeterminantNode data = new ElementDeterminantNode;
                        data.row = n-i-1;
                        data.column = j;
                        data.determinant = matrix[n-i-1][j];
                        data.suborder = i+1;
                        DetArray[n-1] = data;
                else{
                        ElementDeterminantNode data = new ElementDeterminantNode;
                        data.row = n-i-1;
                        data.column = j;
                        for( int x=0; x<i; i++ ){ //as the order increases the sum of the terms
increases
                                a = n-i;
                                b = j+x;
                                data.determinant += Math.Pow(-1, a+b) *
matrix[a][b]*lookUp(DetArray[], a, b);
                        }//end for loop
                        data.suborder = i+1;
                Temp[n-1] = data;

            }//end if
        } //end do while
}//endWhile

Method LookUp(ElementDeterminantNode[], a, b){
        //search for the determinant of the element in row a and column b
        Loop over the input array
            if data.row == a and data.column == b{
                return data.determinant;
            }
        endloop

}
```

Initially the list or array will contain n determinants of the minor matrices of order 1. Then using this information, we calculate the determinants of minor matrices of order 2 and when each is calculated we introduce this into the linked list/array. When all the determinants of the minor matrices of order 2 are calculated then we remove all the 1st order determinants from the linked list. Now we proceed to calculate the 3rd order minor matrices' determinants. When all of them are done, the second order determinants are removed from the list. This process is repeated until we find the determinants of sub matrices of order (n-1). From this point on it is just multiplying by the element and the determinants of its minor and its summation to get the determinant of the input matrix.

Even with iterations we end up doing n! operations. Ignoring the lookup method listed in the pseudocode, for each determinant calculation of submatrix (n-1), we would end up doing (n-2) operations { element * determinant and its sum } and so on and so forth and this will lead us to (n-1)! operations. Since there are n elements, then we end up with n * (n-1)! operations. Therefore, iteratively as well, we would see n! operations leading to O(n!) runtime if we chose to go with the formula listed above.

3. Learnings:

In this project it is clear for the formula given, recursion is the best approach although as the order of the matrix increases the program starts taking longer and longer time with high nonlinear behavior. I believe there are other algorithms like Gaussian row reductions etc. which help us calculate the determinants much faster than O(n!). Given a problem it becomes pertinent to look at alternate methods when we find that the runtime degrades for higher input sets. When we deal with massive data sets with high number of co-efficient matrices, recursion may prove to be counter-productive.

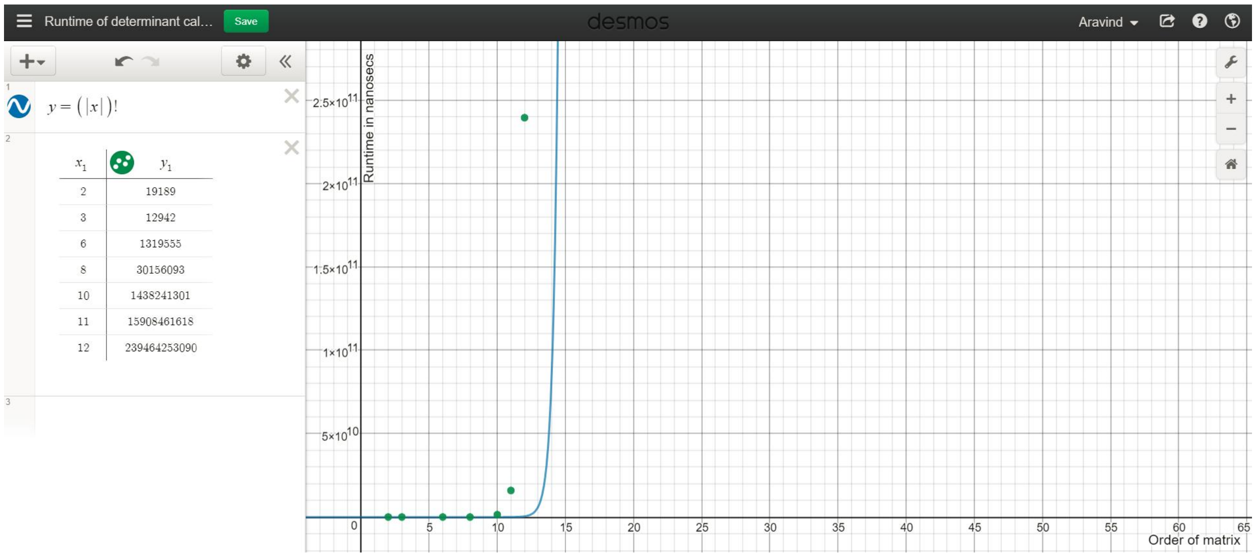
4. Measurements done on some of the methods:

A class utility has been developed to measure the runtime of the algorithms. The class `MeasureDeterminantCalcRun`_does runtime measurements. It resides in com.jhu.ds.matrix.test package.

Size	**Time(nanos)**
2	19189
3	12942
6	1319555
8	30156093
10	1438241301
11	15908461618
12	2.39464E+11

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

When these measurements are plotted they display a highly nonlinear trend as can be seen below (fitting not done). When I tried to calculate the determinant for a 13*13 matrix, it was taking too long. As is explained with the algorithm analysis, the plotted points show an O(n!) runtime. A graph of the factorial function is also plotted alongside to show the trend.
The graphs are plotted by using the free utility from the web desmos.com/calculator



$y = (|x|)!$

x_1	y_1
2	19189
3	12942
6	1319555
8	30156093
10	1438241301
11	15908461618
12	239464253090

Figure 1 Point plot of matrix determinant calculation using recursion