Kevin (Feng Yong) Chen
JHU Email: fchen29@jhu.edu
Student ID: FE4A8B

Lab 2: Analysis

Data Structure and Implementation

Description of data structure and justification

All of the matrices used in this lab were represented using 2D arrays. 2D arrays are the only data structures used in this lab. A 2D array is an appropriate data structure for this lab due to several reasons. Firstly, an array takes less space when storing primitive values such as integers and doubles compared to other structures such as a linked list. Therefore, spatially it is more efficient. Secondly, an array allows for random access to its elements. This makes processing time faster and coding easier when accessing elements in a matrix and when constructing sub matrices. For example, when computing a minor using an array, a new value could be directly inserted into a specific location that is empty in the array without having to go through any other elements prior to it. Thus, having direct access to a location in the array makes processing time faster and coding simpler. Similarly, the determinant algorithm used in this lab allows coders to choose any row or column in a matrix A to compute the determinant of A (see ALT3.2.6). Using a matrix constructed with linked lists, the system would have to traverse through other rows or columns prior to reaching the chosen rows or column, respectively. This, again, consumes time and makes the program slower. Lastly, although a 1D array could technically be used to represent a matrix, I think a 2D array is a much better representation of a matrix since it is conceptually more similar to a matrix – matrix and 2D arrays are both 2 dimensional and both contain row and column information. Thus it is more intuitive to use a 2D array to implement a matrix. Based on the aforementioned reasons, a 2D array seems to be a very appropriate data structure for this lab.

Efficiency

In this section we will discuss the complexity of computing the recursive determinant algorithm. Recall that for matrices with order n>1, the determinant equals the sum of the products:

$$det(a) = \sum_i power(-1, i + j) * a[i, j] * det(minor(a[i, j])), \text{ for any } j$$

Runtime complexity: Based on the above algorithm, a recursive function initially calls itself n times on the first function call, each of the recursive calls will then lead to n-1 recursive calls in the next recursion level, the number of recursive calls should therefore be:

n*(n-1)*(n-2)*(n-3)*.....*1=n!

refer to the code below:

"double product = Math.pow(-1, ((i + 1) + (j + 1))) * matrix[i][j] * det(minor(matrix, n, i, j), n - 1);"

The above codes were copied from the source codes of my "Determinant" class. In the first function call (i.e. order equals n), the function makes n recursive calls. However, for each recursive call the function needs to construct a minor which takes approximately (n-1)(n-1) amount of time to construct. We could then represent the recurrence relation as:

T(n)= n(T(n-1)+(n-1)(n-1))
= nT(n-1) +n(n-1)(n-1)

n(n-1)(n-1) = the processing time of the function itself
nT(n-1) = the number of recursive calls being made

If we ignore n(n-1)(n-1), we are left with

nT(n-1)
= n(n-1)T(n-2)
=n(n-1)(n-2)T(n-3)
.
.
.
= n!

Therefore, in the best case scenario, we have a runtime complexity of Ω(n!). In reality, the runtime complexity is expected to be higher than n! since the runtime of the function itself also needs to be taken into account and the processing time would be affected by the size of the matrix being calculate.

Spatial complexity: At any given point, the system only needs enough space to store things for one branch of the recursion since that space can be re-used once the branch is done executing. That means the system needs space for one function call on recursive level n, one on level n-1, one on level n-2... etc, there are n levels. Each recursive call needs to store a matrix in itself and the matrix takes (n-i)(n-i) space where i represents the depth of the recursion with the very first function call being i=0 (i.e. n^2 for first function call, (n-1)^2 for the next level then (n-2)^2 for the next level …. Etc). Thus, the amount of space needed for this recursive algorithm is:

$$\sum_{i=0}^{n}(n-i)(n-i)$$

However, if we are simply considering the upper bound of the algorithm we could simplify (n-i) to n and the worst case runtime then becomes O(n^3) – we know that in the worst case, the algorithm given in this lab will not exceed n^3.

Appropriateness of Approach

A different approach

Although the algorithm for finding the determinant of a square matrix seems naturally recursive, it is possible to implement the algorithm using an iterative approach. This could be done by using a while loop and a linked implementation of the stack data structure. The nodes used in the stack needs to be structured such that it contains a 2D matrix field to represent a matrix and a multiplier field to represent the ***"Math.pow(-1, ((i + 1) + (j + 1))) * matrix[i][j]"*** part of the algorithm. At each iteration, the while loop will create a minor of the matrix that is at the top of the stack and push that minor onto the stack so that it becomes the new top. The while loop will keep doing this until it detects a minor with n=1. In such case it will start returning values back to the matrices underneath the top. An iterative algorithm for a determinant will look something like what is shown below:

```
public class Node
        double[][] matrix;
        int j;
        int n;
        int count;
        int multiplier;
```

```
        int sum

        public Node (double [][] matrix, n, multiplier)
                this.matrix=matrix
                this.n=n
                this.j=0
                this.count=0
                this.multiplier=multiplier
                sum=0

public class Determinant
        public double[][] minor (double[][] matrix, int n, int i, int j)
                method is same as indicated in my source code

        public double det(double [][] matrix, int n)
                int i=0    // method is always going to choose the first row of a matrix
                int returnValue;
                stack callStack= new stack
                Node initial=new Node (matrix, n, 1)
                callStack.push(initial)

                while(!callStack.isEmpty)
                        Node temp=callStack.pop()
                        If(temp.n==1)
                                Int multiplier=temp.multiplier
                                Int element=temp.matrix[0][0]
                                returnValue=multiplyer*element

                                if(!callStack.isEmpty)
                                        Node temp2=callStack.pop()
                                        Temp2.sum=temp2.sum+returnValue
                                        Temp2.count++
                                        callStack.push(temp2)
                                        returnValue=0
                                else
                                        return returnValue

                        else if (temp==initial && temp.count==temp.n)
                                return temp.sum
                        else if (temp.count==temp.n)
                                returnValue=temp.sum*temp.multiplier
                                Node temp2=callStack.pop()
                                Temp2.sum=temp2.sum+returnValue
                                Temp2.count++
                                callStack.push(temp2)
                                returnValue=0
                        else
                                submatrix=minor(temp.matrix, temp.n, i, temp.j)
                                int multiplier= power(-1,(i+1)(j+1))*temp.matrix[i][j] )
                                Node newStackFrame=new Node(submatrix, temp.n-1, multiplier)
                                Temp.j++
                                callStack.push(temp)
                                callStack.push(newStackFrame)
```

Efficiency of the iterative approach
The iterative approach illustrated above imitates what happens when the recursive algorithm was being executed. Therefore, its runtime and spatial complexity are similar to the recursive approach

In terms of runtime complexity, the number of terms involved should be the same as the recursive approach. Similarly, at each order of n, (n-1)(n-1) amount of time is used to create a minor of an element and n minors are created. Therefore the runtime complexity of an iterative approach should be similar to a recursive approach.

The spatial complexity is also similar to the recursive approach and could be represented by:

$$\sum_{i=0}^{n}(n-i)(n-i)$$

A comparison between the iterative approach and the recursive approach
Although both approaches have similar runtime and spatial complexity, I believe that for this lab a recursive approach is a better way to implement the determinant algorithm. My reasons are as follows:

1) As shown above, an iterative approach is likely to require longer and more complicated codes. It may also require the coder to create more classes to support the determinant method. For example, to actually implement the suggested iterative approach noted above, one would have to create a node class declaring all of the class variables noted above and also create a stack class that uses such nodes. This requires more coding and is more time consuming.
2) On the other hand, the codes for a recursive approach are much shorter and more simplistic. When a coder can write simpler, more elegant codes, I believe that reduces the risk of making careless mistakes.
3) The algorithm is naturally more recursive than iterative as it involves expressing a higher order, more complex matrix in terms of its simpler form. It also provides a base case (i.e. n=1) which allows the system to eventually solve the problem using that base case.

Since the runtime complexity and spatial complexity of an iterative approach is not necessarily better than a recursive approach and a recursive approach allows for a more simplistic and elegant solution, I believe there are more benefits in using a recursive solution compared to an iterative solution for this lab.

Description and Justification of Design
In this lab I only created 2 classes:

1) The main class which process matrices in an input text file and prints the results onto an output text file
2) The determinant class which calculates the determinants of matrices

Since the requirement of this lab is to create a program that reads and print out matrices along with their determinants, I think having 2 classes is sufficient. It is not necessary to break my codes down further into more classes as they are already grouped based on functionality – one class is responsible for determinant calculation, the other acts as a coordinator that receives input, sends it to the determinant class and prints the calculated results onto an output file.

By grouping methods based on functionality, I was able to test a function and make sure it operates properly before moving on to develop the next function. For example, I was able to test and make sure that all of the matrices from the input files are read properly and that various error handling

measures were tested using extreme test cases. By testing the input/output function extensively I was able to detect many potential errors and write codes that protect the user from these errors. For example, through extensive testing, I was able to detect errors such as

- matrices that are not squares (i.e. rows are longer than columns or the opposite),
- Input files having empty lines in between data
- Input files having non-numeric values in the matrices.

These errors were then addressed in my codes and handled appropriately.

There are 2 methods written in the determinant class – a minor method, which calculates the minor of an element, and a determinant method, det, which calculates the determinant of a matrix. Writing minor as a separate method from the det method allows for 2 things:

1) It breaks the codes down so that they are easier for debugging.
   - The minor method was developed and tested first to ensure that it functions properly
   - The det method was subsequently developed and so if there was an error, I could be sure that the problem stems from the det method itself and not from the minor method.
2) It declutters the codes in the det method and makes it easier to read for others. Decluttering the codes also reduces the risk for me to make coding mistake since the codes are simplified.

Lessons From This Assignment

1) In this lab my understanding of spatial complexity has greatly improved through analyzing the space needed for the determinant algorithm
2) I gained a better understand of matrices and learned new concepts such as cofactors/minor and determinants. Through this lab I was able to learn a little bit about linear algebra which I have never learned before. I believe learning about linear algebra will be beneficial to me as a programmer because it is now being applied in many important areas such as machine learning, artificial intelligence and computer vision
3) I also gained insights in how a recursive approach could be converted to an iterative implementation and how a stack could be used in this process. This is useful in the future when I am considering different approaches to solve a problem.
4) I believe reading and printing out the matrices was one of the challenges in this lab. By overcoming this challenge, I became better at using I/O tools and I become more aware of the problems that could occur when scanning data from a text file. I believe this knowledge is valuable to a programmer.

Enhancements

- Although the input data in this lab are presumed to be all integers, I enhanced the program so that it can also compute matrices with values that are doubles. Test cases have been created to test this function.
- When an error occurs, the output will show the user what the error line is so that it is easier for the user to locate the error in the input and fix it.
- The program recognizes when the number of rows in a matrix are greater than the order of the matrix, n, and tells the user that the number of rows are greater than n
- The program recognizes when the number of rows in a matrix are less than the order of the matrix, n, and tells the user that the number of rows are less than n
- The program recognizes when the number of columns in a row does not match with the order of the matrix, and tells the user that the number of columns does not match with n.
- Added in various labels to make the output format more user friendly.

Note. Different input files are created to test whether the program handles various errors properly

Room For Improvement

- I could simplify my code so that it is easier for others to read. I should also try to shorten my codes so that it is not so lengthy.