Lab 3 Analysis – Falko Noe

Design Considerations:

Iteration:

I decided to calculate the determinant using an iterative method, namely row reduction, rather than the recursive method in the previous lab. After doing some surface level analysis of the speed complexity for the previous lab, I decided that calculating the determinant via this method, though more involved from a coding perspective, would result in a much faster run time than the recursive method. This proved itself correct for the most part, as evidenced by the complexity analysis in a later part of this analysis, which asserts that row reduction runs in polynomial time. This is a significant improvement over the factorial time witnessed in the previous lab.

Linked Structure:

The design of the Linked Structure is a uni-directional matrix in two dimensions with header Nodes that takes advantage of the fact that information flow in calculating the determinant via conversion to row echelon form occurs in only one of two directions, down or right. To this end, I decided to write the base *Node* class with only three fields: *Node.down*, *Node.right*, *Node.datum*. The *down* and *right* data fields link to Node that is in the same column position a row lower and to the Node immediately to the right of the Node, respectively. Access to the MatrixList is achieved through a one-dimensional array of Nodes, which gives the program access to the beginning of each row. Since information flow is from left to right and up to down, an additional header array for the columns would have been unnecessary: Since one must iterate through the rows anyway, there are no speed cost savings associated with a column access-point.

Although the inclusion of a header array is not strictly necessary, it has advantages. One advantage is that using an array to store pointers to the beginning of each row simplified the logic involved in adding elements to the matrix, and minimized the need for additional pointers. For instance, starting a new row is as simple as assigning the new Node the corresponding position in the array, and assigning it to the *below* pointer of the Node above, if there is one. In order to gain similar efficiency without an array-based header would involve keeping an additional pointer to the beginning of the row and updating the pointer every time a new row is added. The array header also allowed for row swapping by index, which helped during the debugging phase: As rows are effectively numbered, messages can be printed to stdout which detail exactly which rows were swapped. Access by index is also useful in cases where at least one row sits between the 0^{th} row and the one we want to swap it with, requiring us to reassign the *down* pointers of the in-between row as well: we can simply subtract 1 from the index to access that row. Swapping by index also allows for potential exploits in the data. For example, if we know that we are given matrices where the second-to-last row has a non-zero first element, we could access it and perform the swap without having to iterate through the entire first column. As explained in the complexity section, the additional space used up by the header grows slowly with the size of the input, and is thus worth the tradeoffs.

Fraction Class:

The decision to load the integers from stdin into a custom *Fraction* object within the Nodes instead of simply storing them as integers was informed by two major advantages that such a strategy brings. Since the integers are operated on during conversion to row echelon form and have no guarantee of being divided evenly, the result of such an operation would have had to been stored in the form of a *double*, if not a fraction. However, floating point arithmetic is subject to errors and using this approach could have resulted in incorrect calculation of the determinant. Since hundreds of calculations can occur per matrix, it would have been difficult to pinpoint the origin of such errors. Storing the integer as a fraction, with a numerator part and a denominator part, ensured that no information was lost throughout the conversion process. The second reason for using a custom *Fraction* class is that using a Fraction made debugging significantly simpler: Fractions are easier to calculate by hand, which made verification of my algorithm significantly faster.

However, using a Fraction class also necessitated the inclusion of a *reduce* method to mitigate the possibility of integer overflow. Though matrices of size 6x6 and lower tend to not have any issues of this type, operations on values in larger matrices frequently resulted in such integer overflow, despite the $2^{32} - 1$ maximum value for the *int* primitive in Java. This became apparent because larger matrices repeatedly returned "0" for the value of the determinant, regardless of the actual value that was expected. For this reason, a *reduce* method was necessary. This method reduces the fractions after every operation down to the lowest common divisor and allows the determinant of large matrices to be calculated without having to worry about integer overflow affecting the final answer.

Speed Complexity:

The work of calculating the determinant is composed of three main steps in the worst case: 1) Swapping out the top row for a row in which the first element is not zero, 2) Converting the matrix into row echelon form, 3) Calculating the determinant by forming the product of the top-left-to-bottom-right diagonal.

If the top row's first element is 0, we must find a row below it which has a non-zero integer value as its first element and swap it with the top row. This process has two parts: The first part involves iterating down the rows to find a row which meets the necessary condition. In the worst case, in which the last row is the only row in the matrix which meets the non-zero requirement, we must iterate across all but the first rows in order to find the row we are looking for. This runs in $k - 1$, where k is the matrix's height. Our work is not done. The rows must now be swapped, a process that involves iterating across the rows' values and swapping the elements' pointers. This iteration is repeated for however many items exist in the rows. In all cases, this will be k, the matrix's width. Combining the speed complexity for these two steps is additive, since they occur one after the other. The speed complexity of this step in terms of O() is therefore $O(2k - 1)$.

The second step, converting the matrix to row echelon form, is the most computationally involved. As with the last step, we are effectively iterating down the left-top-to-bottom-right diagonal. Except in this case, we must perform additional work for every element in the

diagonal. Indeed, for each element we perform a series of operations which reduce the elements located in the same column but below the current element to 0. This operation involves finding the multiplicated factor *f* such that $element_{curr} - f \times element_{diag} = 0$, and performing this operation across the entire row, starting at the column index at which the diagonal element is located. For the very top left, this means iterating across *k – 1* rows and *k* elements in each row. Once we are done reducing this column, we advance down the diagonal to the next element, which will be located one row down and one column right from the previous element. Therefore, for this second diagonal value, we must only perform the aforementioned operation on *k – 2* rows and *k – 1* elements. This process continues until the value in the bottom-most row and right-most column, upon which we must perform no operation. This relationship can be easily translated into a sum expression: $\sum_1^k i \times (i-1)$. This is a geometric relationship which can be expressed in O() in terms of k as the following equation: $\mathrm{O}\left(\frac{1}{3}(k-1)(k)(k+1)\right)$.

Dimension length [k]	# of elements looked at
1	0
2	2
3	8
4	20
5	40

The last step, calculating the diagonal, is the simplest. As with step 2, one traverses the top-left-to-bottom-right diagonal. This diagonal is *k* items long: we traverse across all rows from top to bottom and all columns from left to right. As we iterate over the elements, we retrieve the values and form the product with the product of the preceding elements. In the worst case, we must iterate over all *k* elements. The cost in terms of O() is therefore simply O(*k*).

As these three steps occur independently of one another, one after the other, we can add the results of each step to get the overall speed complexity involved in calculating the determinant: $\left(2k - 1 + k + \frac{1}{3}(k-1)(k)(k+1)\right) = 3k - 1 + \frac{k^3}{3} - \frac{k}{3} = \frac{k^3}{3} + \frac{8k}{3} - 1$. Therefore, the speed complexity in the worst case is $\mathrm{O}\left(\frac{k^3}{3} + \frac{8k}{3} - 1\right)$. Since $\sqrt{n} = k$, this can be expressed in terms of *n*, $\mathrm{O}\left(\frac{n^{\frac{3}{2}}}{3} + \frac{8\sqrt{n}}{3} - 1\right)$.

The inclusion of a *reduce* method complicates the analysis of the runtime. Though the subtraction, multiplication, and division operations are constant time operations by themselves, the recursive reduce method generates an additional amount of overhead that does not necessarily scale with the size of the matrix, but rather with the types of integer values that are operated on inside the matrix. For example, while a matrix comprising of only single-digit even numbers would have little overhead, a matrix comprised of mostly large prime numbers would take significantly longer to process, due to the fact that the *GCD* of every prime number is itself and 1. In the worst case, the value is of the form $\frac{odd\ \#}{2}$, i.e. an odd number being divided by 2 which cannot be reduce further, in which case we must make approximately $\left|\frac{odd\ \#}{2}\right| + 1$ (from

here on "m") recursive calls. We do this work to arrive at an answer despite the fact that we do not actually reduce the Fraction, and thus does nothing to lower the complexity of the problem. In the worst case, all values in the matrix are of this type. For the row reductive process, we perform two reductions per element (one for the multiplication and one for the subtraction) and one per row (for calculating the scalar multiple). This ultimately yields: $2m\left(\frac{1}{3}(k-1)(k)(k+1)\right)+m\left(\frac{1}{2}\times(k^2-k)\right)$. Forming the product of the diagonal also requires k – 1 reductions. Adding these all together yields a speed complexity in terms of O() of $2m\left(\frac{1}{3}(k-1)(k)(k+1)\right)+m\left(\frac{1}{2}\times(k^2-k)\right)+m(k-1)+1+2k-1$.

Space Complexity:

The space that is used at any given time is dictated by the size of the matrix that is being considered. As characters are read in and converted into their respective integer representations, they are added to the matrix currently being processed. The most amount of space that will be used at any given time is therefore directly related to the biggest matrix that exists in the input file. Therefore, in the worst case, the entire input *n* composes one matrix with the dimensions of $\sqrt{n}$ in width and $\sqrt{n}$ in height. In terms of O(), this amounts to O(*n*).

There are additional details that must be considered due to the Linked nature of this data structure. Since every integer is stored in the form of a *Fraction*, which is composed of a numerator and a denominator, every integer we read in will effectively take 2 integers worth of space in the program. We must also account for the pointers involved in the Node object, which take up space in the form of two addresses, one pointing down and one pointing right. Lastly, the inclusion of a static header Node-array ensures that we must add an additional columns-worth of space for each complete matrix, equal to $\sqrt{n}$. Assuming all the mentioned additional space requirements take approximately the same amount of space, then accounting for these factors results in a worst case O() of roughly O($4n+\sqrt{n}$). Dropping the constants simplifies the expression to O$\left(n+\sqrt{n}\right)$.

Considering the space required by the recursive method *reduce* adds space equal to the maximum number of recursive calls that is made to *reduce*. This in turn again depends on the types of data in the matrix. The most of amount of space we will use simultaneously is therefore simply the aforementioned formula plus the number of recursive calls are made to the least optimal Fraction, which will be of form $\left|\frac{max\ odd\ \#}{2}\right|+1$. Adding this to formula above yields O$\left(n+\sqrt{n}+m\right)$.

Enhancements

Beyond the inherent speed boost offered by iterative row reduction, I attempted to build in a number of further efficiencies into my solution.

Transforming a matrix into row echelon form lends itself to several early-outs for computing the determinant. The first occurs while looking for rows against which to swap. If we fall off the

matrix, that is to say we don't find any non-zero values in the first column, we can simply return 0 since a matrix with an entire row or column of only 0s has a determinant equal to 0. The speed complexity of such a matrix will run in $\sqrt{m}$, where *m* are the number of elements in the matrix.

The second early-out occurs during conversion of the matrix into row echelon form. If we encounter an element in the diagonal which is equal to 0, we can simply return 0 as the value of the matrix's determinant. This is because the product of the diagonal will necessarily be 0 if any of the elements along it have a value equal to 0.

Another time-saving exploit that we can take advantage of during the conversion of the matrix into row echelon form is when the first element in a given row at the column starting point (the column of the value in the diagonal currently being compared against) is equal to 0. Since the scalar multiple in the expression $R_{curr} - f * R_{diag}$ will also equal 0, the value R_{curr} will remain unchanged. This exploit requires an additional comparison, but the payoff is large because it means we do not need to look at the rest of the values in the row and can instead continue on to the next row, if one exists. Matrices that contain a great number of 0 values in their lower halves benefit the most from this exploit: matrices such as the 16x16 in the input file execute fast despite their large size.

Lab2 Solution vs Lab3 Solution

As stated above, the iterative row reduction solution implemented in this lab is significantly more efficient than the recursive one: the iterative solution runs in polynomial time, while the recursive solution runs in factorial time. Though the iterative solution has the added overhead of the recursive *reduce* method to deal with, the solution will nonetheless be significantly faster than the recursive one for most inputs. From a performance perspective, the lab3 solution is superior, specifically with regard to large input sizes. For example, the 16x16 input matrix runs almost as fast as the smaller matrices in the input file for the iterative solution. The recursive solution, on the other hand, did not complete its calculation within the 2 hour time frame I gave it to run. For real-world applications, the iterative solution is therefore a more realistic approach to calculating the determinant.

From a design perspective, the lab2 solution is significantly simpler. The logic of the recursive call, though perhaps more abstract, is significantly easier to write and easier to understand. It required less debugging and requires no special handling in its naïve form. The iterative solution, on the other hand, required significantly more thought to implement. Though the logic itself was simple enough, certain issues such as the aforementioned integer overflow, came unexpected and had to be worked around.

From a data structures perspective, the array implementation could have been used just as well in the iterative solution with only minor tweaking necessary. Instead of navigating through the matrix with *next* and *down* pointers, one would simply have to increase the row or column index as necessary and use the 2-D array's random access to get the next value. Using a Linked implementation for the recursive solution would have also been trivial. The main difference would have been in calculating the minor, which would have required building an additional LinkedMatrix for each call to minor. Otherwise, the logic remains the same.

What I Learned

One big take away from implementing an iterative row reduction was realizing that integer overflow can sneak its way into applications if one is not careful about the types of integer operations that are performed. I had not considered that reaching the 2^{32} limit would occur under most normal applications. However, as previously mentioned, this was a common occurrence on matrices exceeding 6x6 prior to inclusion of a *reduce* method. After comparing my iterative solution to the recursive one from lab2, I noticed that the recursive solution was also susceptible to integer overflow. Running the 7x7 matrix through both programs resulted in the correct answer in my iterative solution, but resulted in the wrong one for my recursive one. This is not surprising, since the recursive solution ultimately relies heavily on multiplication: minors are multiplied by the value at a given row/column index. If large enough values are calculated for subproblem, the 2^{32} integer limit can be reached very quickly. For instance, the product of multiplying 256 x 256 x 256 x 256 x 256 exceeds the maximum int value by three orders of magnitude. Row reduction using Fractions that are reduced by the greatest common factor run into this issue less frequently, since all terms are kept the lowest they can possible be. As stated above, this does come with a cost however: the *gcd* method is an expensive one, especially if either the denominator or numerator are prime numbers.

Future Improvements

In future versions of this program, I would like to implement the *reduction* method iteratively. Size of the integer permitting, this would allow me to process much larger numbers and would ensure that the stack depth of Java would not be exceeded should large prime numbers be found in the input file.

Though I was happy with my inclusion of a header Node, I believe it would be possible to implement the MatrixList without one. It would be possible to have a single entry-point into the List, from the very first Node for example. Beyond the entry node, one would simply have to keep track of certain pieces of information, such as the current row, using locally scoped variables. This would allow me to remove the Node array and reduce my memory requirement by a single column worth of integers. Though this is not a huge gain in space, it would be worth it in cases where memory is the limiting factor.