Implementation Description: My implementation is based around two structures, an Entry structure and a Matrix structure. The Entry structure contains a pair of integers used to represent the coordinates of that entry, and the value at the entry itself. The matrix structure is a linked list of Entry objects and an integer that contains the expected row size of the matrix. To load the matrix, I modified the GetInput() function from Stepik Exercise 2.3. I modified it so that it returned a matrix object instead of a vector, and it stored only the non-zero values in a list, rather than all the values into a vector. After the matrix is loaded in, I pass it to the Determinant function. The base case of the determinant function is when the expected row size of the matrix is 1, meaning that it is a 1x1 matrix. A 1x1 matrix’s determinant is equal to its single entry, so calculating the determinant of a matrix object with an expected row size of 1 is simply checking if the list is not empty and then returning the value at the front of the list. If the list is empty, zero is returned, because that means that the entry at the location should have been a zero matrix. The recursive step works by iterating through the top row of the matrix and taking the minor of the matrix relative to each entry in the top row. The determinant is then incremented by (-1)i+j \* a[i, j] \* det(minor(a[i, j]), as given in the project specifications, for each entry that is on the top row of the matrix. The minor matrix is calculated by passing in the original matrix and the coordinates of the entry to be removed. The list from the original matrix is then iterated through and only the entries that do not have the same x-value or y-value (are not in the same row or column) are added to a new list. I then create a Matrix object with this new list and an expected row size one less than the original expected row size, as each time a minor is created one column and row is removed, so the row size will decrease by one.

Why I Chose It: This implementation evolved around the first structure I could think of to calculate the matrix determinant, which was the Entry structure. I knew that because I was skipping over zero values, I would need some way to keep track of where the non-zero values originally were in the matrix. The easiest way I could think of to do this was to create an object that could store both the value at the entry and a pair of coordinates that represented how far away from the origin that value was, with the origin being the top-left of the matrix. I got the coordinates originally from loading the matrix in, as my LoadMatrix() function uses a for loop and a currSize counter. The for loop variable acted as the y, as it would increment for each line, and the currSize variable acted as my x, as it would increment for each number in the line. If the entry was equal to zero, then no Entry structure would be created, but the numbers would still increment correctly. I originally was just going to use a linked list and not have a matrix structure, but that lead to problems with using the size of the linked list to determine when to find the minor of the matrix, as the matrix could actually be larger than its one non-zero element. So I decided to create a structure that held the non-zero values as well as an expected row size, and then used this expected size to determine when to get the minor.

What I Learned/Most Difficult Part: This assignment gave me a better understanding of how recursion can solve a problem by breaking it down piece by piece. For example, the most difficult part of the assignment to me was figuring out how to change the coordinates of each entry in the minor to match the new origin. At first I tried to base it off the x coordinate of the entry that was being removed, but that didn’t work. I finally realized that I only needed to decrement each coordinate value by one each time, as only one row and column was being removed. Additionally, the x-value only needed to be decremented if the entry to be added was to the right of the column being removed, as otherwise its x-coordinate would be unaffected. However, because the top row was always removed, the y-coordinate should always be decremented by one. Solving this part of assignment gave me a better understanding of what makes recursion special, as recursion breaks problems down by making the problem little (or a lot) smaller each time it is called until we reach a base case that is easily solvable. This conflicted with how I normally write projects which is to try to solve everything at once using the data I have. However, some problems are simply too complex to be solved this easily and must be broken down until they are simple enough.

Computation Complexity:

list<Entry> minorList; // O(1)   
**auto** iter = matrix.matrixList\_.begin(); // O(1)  
**while** (iter != matrix.matrixList\_.end()) // O(n)  
{  
 **bool** insert = **true**; // O(1)  
 **if** (iter.**operator**\*().coords.first == removed.first || **int** x = iter.**operator**\*().coords.second; == removed.second) // O(1)  
 insert = **false**; // O(1)  
 **if** (insert) // O(1)  
 {  
 **int** y = iter.**operator**\*().coordinates.first - 1; // O(1)  
 **int** x = iter.**operator**\*().coordinates.second; // O(1)  
 **if** (x > removed.second) // O(1)  
 x = x - 1; // O(1)  
 Entry inserted(y, x, iter.**operator**\*().entry); // O(1)  
 minorList.push\_back(inserted); // O(1)  
 }  
 iter++; // O(1)  
}  
Matrix minor(minorList, matrix.expectedRowSize\_ - 1); // O(1)  
**return** minor; // O(1)

Total Complexity: O(n)

Each of the operations is O(1) except the while loop, which loops through the entire list. Thus the computation complexity is O(n), with n being the number of non-zero entries in the matrix.

**int** Determinant(Matrix matrix)  
{  
 **int** determinant = 0; // O(1)  
 **if** (matrix.expectedRowSize\_ > matrix.matrixList\_.size()) // O(1)  
 **return** 0; // O(1)  
 **if** (matrix.expectedRowSize\_ > 1) // O(1)  
 {  
 **auto** iter = matrix.matrixList\_.begin(); // O(1)  
 **while** (iter.**operator**\*().coordinates.first == 0) // O(sqrt(n))  
 {  
 **int** y = iter.**operator**\*().coordinates.first; // O(1)  
 **int** x = iter.**operator**\*().coordinates.second; // O(1)  
 pair<**int**, **int**> remove(y, x); // O(1)  
 Matrix minor = GetMinor(matrix, remove); // O(n)  
 **if** ((x + y) % 2 == 0)  
 determinant += iter.**operator**\*().entry \* Determinant(minor); // O(1)  
 **else** determinant -= iter.**operator**\*().entry \* Determinant(minor); // O(1)  
 iter++;  
 }  
 }  
 **else** {  
 **if** (matrix.matrixList\_.empty()) // O(1)  
 **return** 0;  
 **else  
 return** matrix.matrixList\_.front().entry; // O(1)  
 }  
 **return** determinant; // O(1)  
}

Total Complexity: )

The complexity of the determinant calculation is ), where n is the number of non-zero entries in the matrix. The while loop goes through the entire top row, and because this is a square matrix this is equal to of the total number of entries. Each time the loop is traversed, GetMinor() is called, and GetMinor has a complexity of n, as seen above. Thus one call of the recursive function has a complexity of ), and the number of recursive calls is done times, as it is done across the top row of the matrix. Thus the total computation complexity is

).