Algebra Calculator

Lucas Diffey, Thomas Hood, Joseph Gallagher
Fontbonne University

Abstract

We have developed a simple CAS that is capable of performing a few different tasks: solving complex equations and retrieving values from a mathematical statement, and with minor edits to the source code, would also be able to graph functions. When developing the program our team decided that lower-level mathematics would be our primary audience, so much of the functionality of the program is designed to serve their needs. Our team created a real-world use case for the program as serving as a learning tool for students and teachers; the program has a built-in algorithm for printing out step-by-step solutions for the math equations that it solves, allowing students to understand visually how to work through more challenging problems.

Background

Logic-based mathematics is a cornerstone in computer programming, so we decided to create a program with an emphasized focus on mathematics. As many students know, algebra is a notoriously difficult class. Our program was designed with students and teachers in mind, both high school and college audiences and specifically in algebra courses, to solve simple systems or functions of equality. The architecture of the program allows more experienced or ambitious students and teachers to modify the program in a way that meets their desired performance needs. Programs like PhotoMath, Mathway and Desmos are all existing programs and proof od concept that this program meets a real-world need. Our team is still considering a possible port to Android/IOS phones, but as of right now there is no official scheduled release.

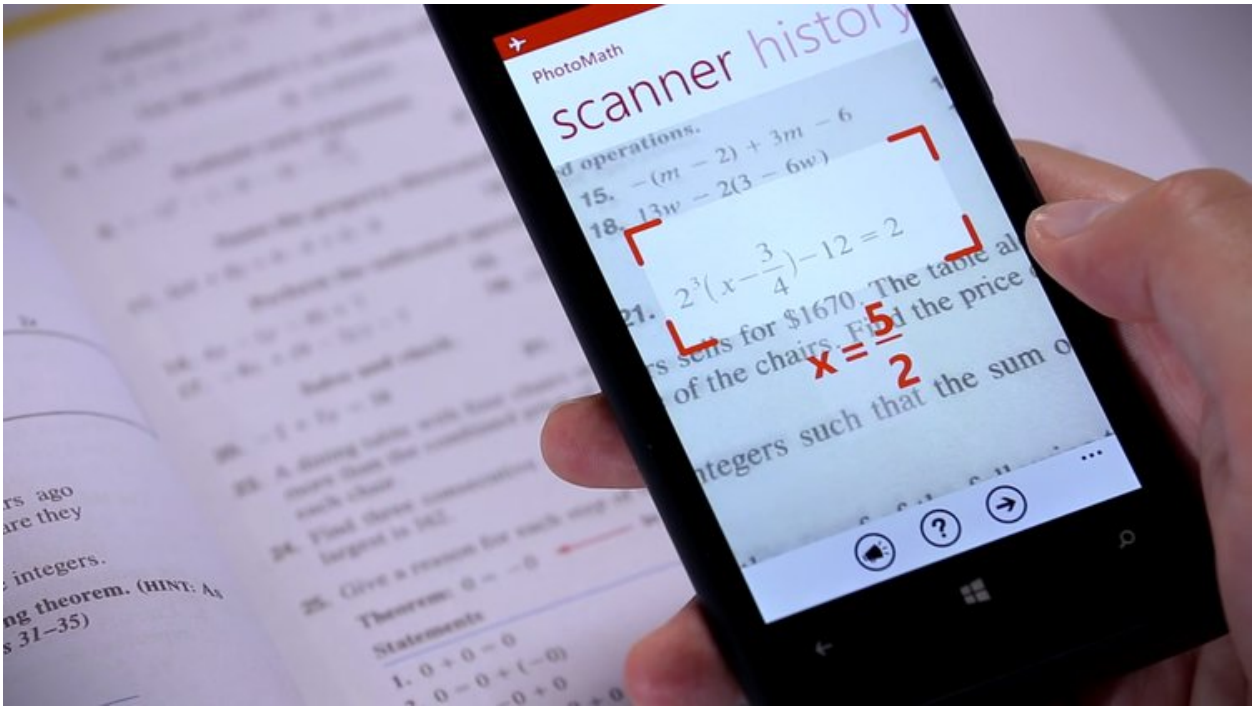

Figure 1. A stock image from PhotoMath demonstrating its ability to solve algebra problems using a picture stream of the problem. PhotoMath has been a great inspiration to our team and potential developments for mobile devices. Business Insider. *PhotoMath*. Apr, 2018. Digital.

This program, unlike other programs, has the capability to display the solution and associated steps to solve the problem. After the solution is provided and the user moves onto his next question, there is an included setting to download all questions and solutions used during the session as text-file. Our program will allow users to save their work anywhere on their computer for future reference at any time.

Implementation of Stacks

Overview:

In computer science, stacks are abstract data types that hold a collection of similarly-typed objects. What is notable about stacks, as opposed to other abstract data type collections, is that they are a linear data structure and operate using a method known as FILO (or LIFO), which is First in Last Out (Last In First Out). This means that objects or data that you add into a stack are only retrievable from the "top" or front of the stack and also newly added data will be retrieved before data added earlier. In a sense, stacks operate similar to a "stack" in real life; imagine a stack of books lying on a tabletop - you cannot access a book from the bottom or middle until you first remove the ones from on top of it. When using stacks, the two main methods used to operate the data structure are the push and pop methods, which respectively add and remove data from the structure.

Architecture:

Typical stacks are constructed as an array of computer memory with a fixed origin and variable size, which is initially set to zero. If the origin of the stack is at an arbitrary address 500, then the stack must not be incremented outside the allotted address pool; doing so could result in a stack underflow or overflow, which means that the stack is attempting to access unassigned computer memory. Stacks operate using pointers which will "point" to the most recently referenced location on the stack, and in the case of an "empty" stack (size of zero) the pointer will point to the origin of the stack; likewise, the previously mentioned push and pop methods operate using the stack pointers. Pushing an item onto the stack adjusts the stack pointer by the size of the item - either decrementing or incrementing, depending on the direction in which the stack grows in memory - pointing it to the next cell, and copies the new top item to the stack area. Depending again on the exact implementation, at the end of a push operation, the stack pointer may point to the next unused location in the stack, or it may point to the topmost item in the stack. If the stack points to the current topmost item, the stack pointer will be updated before a new item is pushed onto the stack; if it points to the next available location in the stack, it will be updated after the new item is pushed onto the stack. Popping the stack is simply the inverse of pushing; the topmost item in the stack is removed and the stack pointer is updated, in the opposite order of that used in the push operation. Finally, stacks have additional methods such as duplicate, swap, exchange, rotate, and more, but these methods' availability vary from environment to environment. The most important architectural aspect to stacks is their FIFO property which dictates the order of insertion and removal from the structure, making them function similar to "real-world" stacks (Somogyi & Stuckey 1).

Use of Stacks in our Program:

For the implementation of our program we decided to use two stacks to store data entries for algebraic operators and floating-point numbers. The decision to use two stacks will be explained later in the research paper, while this section will focus solely on the use of stacks in our program as opposed to different data structures. For the program's implementation, our group required a data structure that is dynamic and able to grow to accommodate user interactions. Additionally, we wanted to be able to easily reference and use recently entered data in the structure, making stacks the ideal option. The FIFO architecture present in stacks was perfect for the implementation of our program because it allowed us to read in an algebraic expression from the user as a string and then parse it to separate floating-point numbers and operator characters into different stacks and then use the pop method to retrieve an operator from the first stack and then perform it on its corresponding floating-point numbers in the separate stack. While this program could also be implemented using a queue, the algorithm that we created and used to read in user input and parse it was more optimized for a stack. Using a stack in conjunction with our created algorithm allowed us to have a program time complexity of $O(N^2)$, which is not perfect, but does satisfy our predetermined project goal. We found that if we could eliminate our use of nested loops we could further reduce our time complexity to O(N) or O(1). This is primarily because stacks have a constant time complexity when utilizing methods like push, pop, and deletion, which means a majority of our program runtime is occured the nested loop statements.

```java
import java.util.*;

class StackDemo {
  public static void main(String[]args) {
    Stack<String> stack = new Stack<String>();
    stack.push("A");    // Insert "A" in the stack
    stack.push("B");    // Insert "B" in the stack
    stack.push("C");    // Insert "C" in the stack
    stack.push("D");    // Insert "D" in the stack
    System.out.println(stack.peek());    // Prints the top of the stack ("D")
    stack.pop();    // removing the top ("D")
    stack.pop();    // removing the next top ("C")
  }
}
```

Figure 2. A simple code sample of program implementation for a stack. This sample demonstrated the use of the push and pop methods, and prints out the contents of the stack to the console. Java Class Library (JCL). *Stack*. Apr, 2018. Digital.

Single vs. Double Stack:

After trial and error during program implementation, our team ultimately decided to use two stacks, which proved better for managing the operators and subsequently applying them to their corresponding floating-point numbers. While using two stacks did make implementations easier for our group, we did take note that it also resulted in larger space complexity. Ideally, this program would eventually be optimized to a point that our algorithms could work off of just one stack, but as of now the algorithms are designed for two stacks, which is slightly suboptimal. However, it is also worth noting that using two separate stacks did not negatively impact the program time complexity. Ultimately, looking to the future of this program, our team has concluded that reconstructing our algorithm to distinguish operators from operands within a single stack would be the best course of action, however, careful planning and trial runs will be necessary so that the new algorithm does not further increase time complexity.

Computational Algebra

Overview of Computational Algebra:

Computational Algebra, also known as symbolic computation or algebraic computation, is a highly specific area in science that relates to the development and study of algorithms and software for manipulating mathematical expressions and other mathematical objects. To be accurate, computational algebra is actually a subfield of a broader area called scientific computing, but because of the overwhelming specifics and specialization of computational algebra, the two are practiced and studied as separate fields. Additionally, it is noted that scientific computing tends to deal with numerical computations with *approximate* floating point numbers, while computational algebra emphasizes using *exact* computation via variable expressions, symbols, and mathematical manipulation. Software applications that perform these symbolic calculations are called computer algebra systems (CAS); the term system alludes to the complexity of the main applications that include, at least, a method to represent mathematical data in a computer, a user programming language, a dedicated memory manager, a user interface for the input/output of mathematical expressions, and a set of routines to perform usual operations (simplification of expressions, differentiation using chain rule, polynomial factorization, indefinite integration, etc). Computational algebra is widely used in experimental mathematics and formula design for numerical-based programs and languages, and a few unique scientific computations (public key cryptography, non-linear computations, etc.). (Wolfram 1).

Our Program as a Computer Algebra System (CAS):

A computer algebra system (CAS) is any mathematical software with the ability to manipulate mathematical expressions in a way similar to the traditional manual computations of mathematicians and scientists (Wolfram 1). Typically, computer algebra systems may be separated into two classes - specialized and general-purpose. Specialized systems are dedicated to a specific part of mathematics (number theory, group theory, teaching elementary mathematics, etc.). General-purpose systems are designed to be useful to users working in any scientific field that may require the manipulation of mathematical expressions. These requirements are further demonstrated by the following documented criteria for general-purpose computer algebra systems:

Documented Useful Features for CAS

-User interface for entering and displaying formulas
-Simplifier for rewriting expressions and formulas
-Precision-based arithmetic
-Programming language and interpreter
-Memory manager and garbage collector
-Libraries for users and simplifiers

These large amounts of required computer capabilities explains the small number of general-purpose computer algebra systems. The most notable general-purpose computer algebra systems are Axiom, Maxima, Magma, Maple, Mathematica and SageMath. Our program does not hold up to the standards of the previously mentioned programs, but it also is not designed to handle as many expressions and functionalities; our program was designed to fill a specific niche in assisting high school and college algebra students by simplifying and solving basic algebraic problems and displaying the step-by-step procedure. In light of this, our program would best be classified as a specialized computer algebra system. While this program does have many of the capabilities listed above, such as a graphical user interface, programming language and interpreter, simplifier, memory manager and garbage collector, and precision-based arithmetic, it lacks an abundance of libraries for users and simplifiers. At its current stage, this program is only designed to work with basic algebraic inputs and cannot handle variable expressions above an order of 1, trigonometric expressions, or matrices. Additionally, there is no option to successfully graph input functions. However, looking at future developments for this program, these options are implementations our team would like to include in a final commercial or open-source release.

Implementation of Solution in C#

The creation of the Simple Algebra Calculator required optimized and precise algorithm design and implementation. Our team opted to use multiple data structures and code modularity for the finished project. The main data structures used to make this code functional are stacks and lists. Stacks are useful when storing data because of their unique first in, last out (FILO) property. An easy way to think about a stack through a real life example would be a stack of plates. The newest plate gets added to the top of the stack. However, when removing, the top plate must be removed before any of the others can. This data structure helped us to keep track of the order of numbers and operations. One stack is dedicated to holding numbers, and the other holds operators. As mentioned earlier, the project also implements the List data structure as well. A list in C# is a dynamic array, which is almost identical to an ArrayList in Java; there happens to be an ArrayList class in C#, but during prototyping and testing, our team found List to be the optimal data structure. In fact, "The ArrayList class is designed to hold heterogeneous collections of objects. However, it does not always offer the best performance. Instead... for a heterogeneous collection of objects, use List<Object>" (Microsoft 1). Lists do not have set sizes and elements can be added to them very easily. Having a dynamic array is crucial when we do not know how many problems the user will enter and how many steps it will take to solve each problem. Although we did not utilize them, there are many methods additional to the add function; such as remove, sort, clear, and many others (Microsoft 1).

Essentially, our program needed three functions in order to handle the calculation of values. The functions used can be found in the appendix below, and are as follows: Calculate Expression, Order of Operations, and Operation. Calculate Expression is a function that takes as input the expression that the user wants to solve. It then creates a stack for numbers, and a stack for operators. Next, the input string is converted into a character array. Now we can start traversing through the array so that we can figure out the positioning of the numbers and the operators. When the current character is an empty character, we move on to the next character in the array. This serves as the break case because there are lot of white spaces in between the values from the user. Next, the character is checked to see if it is a number between zero and nine. If this is the case, two things happen. The first is that a stringbuilder object is used to create a string of the values. The next is taking that number and pushing it onto the stack so that it may be accessed. Next, we check for an opening parentheses. If its there, it is pushed onto the operator stack to be accessed later to handle the order of operations which is very important. Next of course, we must check for closing parentheses so that an erroneous input does not crash the program. Therefore, we use the peek method on the operator stack to ensure that the top element is the opening parentheses. If so, we push the result of the Operation function to the number stack. The Operation method takes an operator from the stack and the two topmost elements from the number stack and does the calculation. The next step is to handle when the

current element is an operator. Basically if the current character happens to be an operator, then we push onto the numbers stack the result of the Operation. Next, we push that particular operator to the stack. The final check we must make is if the character is a variable. In this case, the user must substitute x for whatever variable they wish to solve. In the future, we would like to make the code more flexible so that any alphabetical character may be entered. First, we must check the case of having a number, an operator, and then x. We then use the location of x to backtrack and find the operator in front of it so that we can hide it. The functions that we implemented to not handle variables; the event handler for the solve button handles the variable. Next, we check for the opposite case - x, an operator, followed by a number. We then do the same thing as before; convert the x and the operator to a blank character so that they do not cause errors in the method. Finally, we push the result of the Operation method one more time to the number stack. Doing so ensures that the calculated values are inserted into the stack so that they can be used later in calculations. Due to the fact that the user can attempt to break our code, we have a catch for generic exceptions so that the application does not crash.

The Order of Operations and Operation methods are just as important as the Calculate Expression method. The Order of Operations method takes in the two topmost values from the operator stack as input and returns a boolean value. If the second operation is a parentheses, or if the first operation is a regular operator, return false. Otherwise, for our default case we return true. The Operation method is a bit more involved. It has a return type of integer and takes in an operation and two values as input. Now, we must figure out the operation to do on the values. If it is multiplication, we multiply the values together and return the product. We create a string that displays that we must multiply the numbers and what the intermediate product is to the textbox on the application. We do the same for division, where we return the quotient of the numbers and the intermediate result, addition where we return the sum and the intermediate result, and subtraction where we return the difference and the intermediate result. All of these results are stored appended to the textbox text using a static string variable. In C#, this is basically a member variable that can be accessed throughout a project - even other classes. Each time an operation occurs, this intermediate step is appended to the textbox to show the line by line results. *Figure 3* below provides a visual for the main algorithm design of the project.

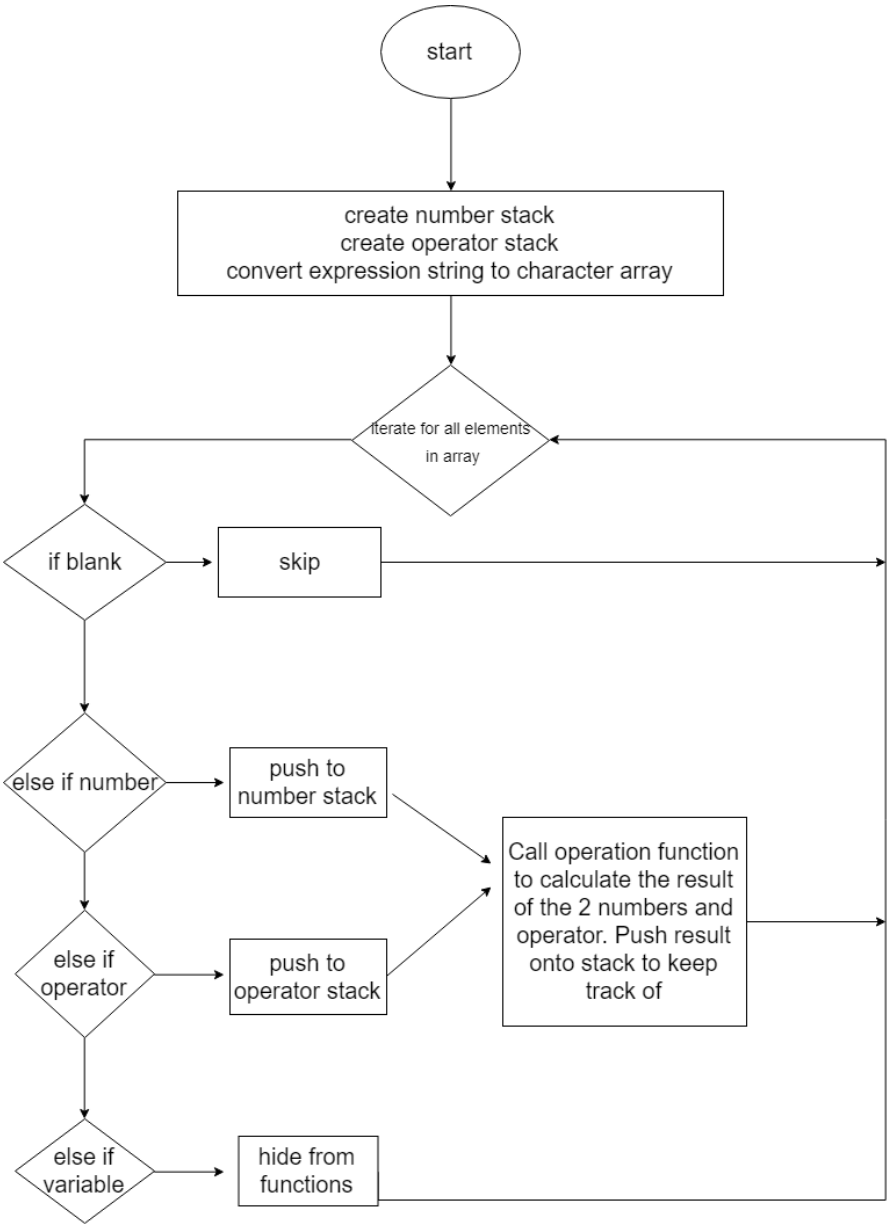

Figure 3. Flowchart for the algorithm that is responsible for calculations.

As mentioned before, C# auto generates event handlers for toolbox items, such as text boxes, buttons, and etc. Since the textbox gets populated with the data when the user clicks the solve button, the event handler for a click of the solve button acts as our 'main function'. In this event handler, we get the resulting expression that is entered by the user. Next, we split the string on the equal sign so that we have a left and right hand side to solve. After getting the two sides of the equation separated, we call the Calculate Expression method on both sides to get the results. Due to the fact that we remove 'x' from the equation, we must now place it back into the expression where it belongs. At this point, we should have a number, an equal sign, followed by another number. To figure out the placement of 'x', we look back at both sides of the equation to

get its positioning. Then, we add the variable back to the string holding the equation. Now that 'x' has been replaced in the string, we must figure out the sign on the side with 'x'. Depending on what that is, we do the opposite operation of whatever value remains on the same side as the variable. As in basic algebra, we must do this operation on both sides. This final result is appended to the textbox and the full text is added to the next element of the list so that it may be saved to a text file. These are the main parts of the code that do the calculations. While there is other code, this is the most important for calculating the result that the user enters - which is the whole purpose of the application.

While all of the members in the group also know Java, we decided to code this application in C# for many good reasons. Java and C# are very similar languages - they are both object-oriented, strongly typed, highly portable languages that have many libraries and packages to help developers. "Like Java, C# co-exists with a runtime environment (like Java's JVM); C# has ways to communicate on the network (like Java's RMI but unlike Java's applets); and several independent technologies are used by both languages (such as XML). Much of C# is the same as Java" (Bishop, Horspool, & Worrall 1). Despite the similarities, C# is leagues ahead of Java in terms of Graphical User Interface (GUI) development. Microsoft Visual Studio allows for drag and drop development for creating forms - whatever you need in your application can be selected from the tool box, dragged onto your form, and dropped wherever it is needed. Therefore, in order to create buttons, textboxes, or even menustrips, no lines of code are required whatsoever. Visual Studio generates that code for the developer automatically. Also, every item in the toolbox has properties. These, like the items themselves, require no lines of code and provide quick and easy access to the settings of the items. For example, by selecting a textbox on a form and opening up the properties about it, a developer can make it read only, have scroll bars, and change the font all by the click of a mouse. Besides making GUIs much more quickly in C#, it also allows for more customization and thus more aesthetic applications for the user. Another advantage with C# is that by double clicking any item that has been added to a form, Visual Studio automatically creates event handlers for interaction with that item. An example of this is a button on the form. If the developer wants a message box to display when a button is clicked, all that has to be done is add that code to the event handler for when the button is clicked and it will be displayed. Although Java has an API called SWING that assists in GUI development and design, it is not as simple as drag and drop development which is provided in C#. Another advantage of using C# is that there is a browser built into Visual Studio that allows developers to download special packages to enhance their programs. For example, a future step for this project may be to graph functions so that students can visually understand functions. There are many packages available that would allow us to implement this helpful feature. Importing packages into Eclipse, which is an IDE for Java, is not nearly as easy. You have to browse outside of the application which runs a higher risk of downloading malicious code. One of the main drawbacks surrounding C# and Visual Studio is that the programs are much larger so they compile more slowly than Java will. While they both sacrifice speed for portability, C# uses

the Just In Time (JIT) compiler and Java uses the Java Virtual Machine (JVM). Both of these compilers are known for lacking speed because they have to convert the byte code into machine code before the program can be executed. While C# is slower the first time you execute the code, our application has a great runtime considering the amount of graphics used to display the GUI. Through experimental studies and asymptotic analysis, we can see that the program actually runs considerably faster after the first execution.

Not only does our implementation solve equations involving basic algebra, it does so while being user friendly and very useful. *Figure 4* below shows the GUI design:

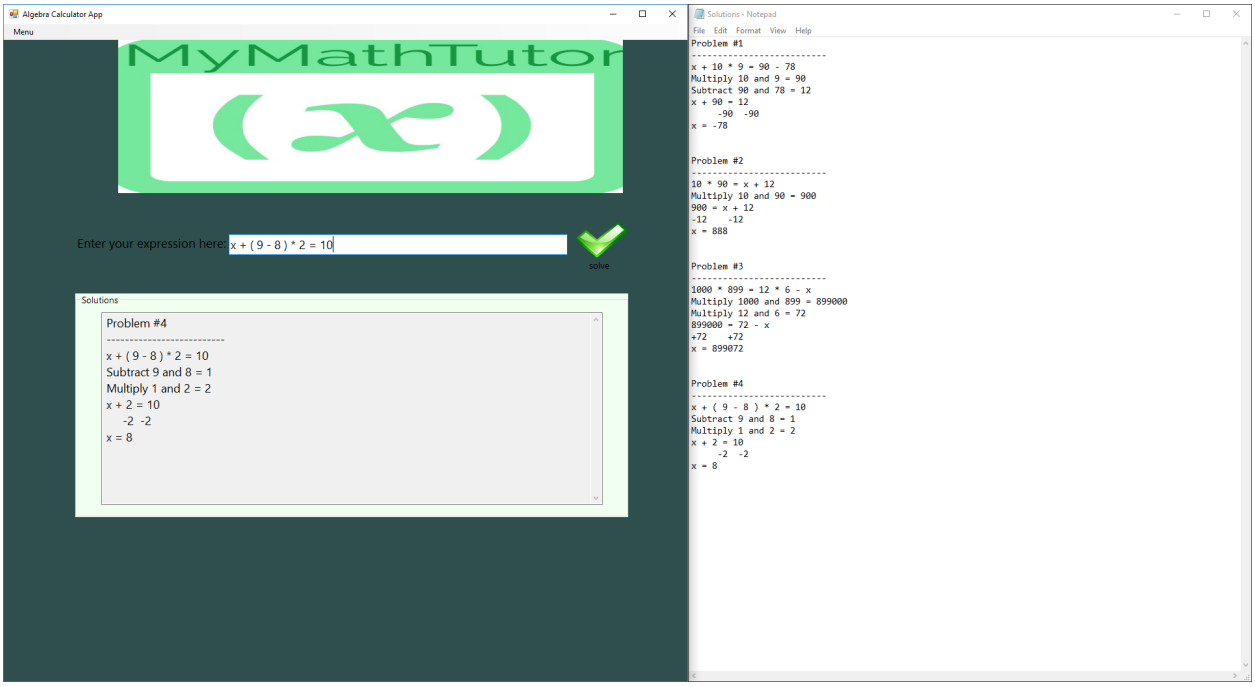

Figure 4. The GUI design and its main functionalities, including saving the problem set to a textfile.

The user has multiple options when using the GUI. The first and most important is entering an equation into the textbox and clicking the solve button in order to generate the solution. The "menu" at the top provides additional functionalities. When "menu" is selected, a drop down list of three actions is made visible: About, Save, and Exit. By selecting about, a window with a description is displayed so new users can understand the GUI. The most important additional action is save. This allows the user to save to a text file all of the problems that they have entered for that problem set. When clicked, a File Explorer Window is displayed and the user can choose to save the file wherever they choose and name it whatever they choose. Everything that is displayed in the textbox as far as the solution is included in the text file. This is an important tool for teachers and students alike so that they may look back at previous homework sets and understand the process of problem solving. The final button under menu is the exit button.

Clicking this will display a window alerting the user that by leaving without saving may result in lost data. The user can either continue exiting or choose to go back to the GUI to save the problem set.

Experimental Studies

Now that we have considered the design of the algorithm and the implementation in C#, we must take a look at the performance of our application in terms of asymptotic analysis and experimental studies on runtime. First, we will look into the asymptotic analysis of our application. To do this, we will consider Big-Oh Notation for both time and space complexity. Big-Oh notation accounts for the upper bound of functions, which is almost like a worst case scenario. "Big-O is used in computer science to estimate the upper-bound of an algorithm's runtime. We assume that big-O is important... to write efficient code" (Parker & Lewis 201). To analyze the time complexity, we must look at the program for key features. There are seven possible time and space complexities: constant where f(n) = c, logarithmic where f(n) = log(n), linear where f(n) = n, linear logarithmic where f(n) = nlog(n), quadratic where f(n) = n^2, cubic where f(n) = n^3, and finally exponential, where f(n) = a^n (a is a constant). In order to analyze the code, we must know a couple of basic rules. Assignments and arithmetic are constant time, and loops are linear time. These basic rules should be all we need to determine the time and space complexities of this program. Since we have nested loops in the Calculate Expression function, we know that the time complexity is O(n^2). Although this is not optimal, this is the largest time complexity that we originally wanted to have. Space complexity is even more simple to determine. We know that the space complexity of the program is O(n) because we use a list (dynamic array) as storage. Since lists and arrays take up contiguous memory, or linear blocks, we know the storage is in fact linear. Therefore, the program could have better space and time complexities, but in order to make the application perform the way we want it to there is no way around it. Sometime in order to increase usability and functionality, performance must be minimized as a sacrifice.

Now that we have analyzed the functions appropriately, we can analyze the code via experimental studies. *Figure 5* below shows the runtimes in nanoseconds and the run number.

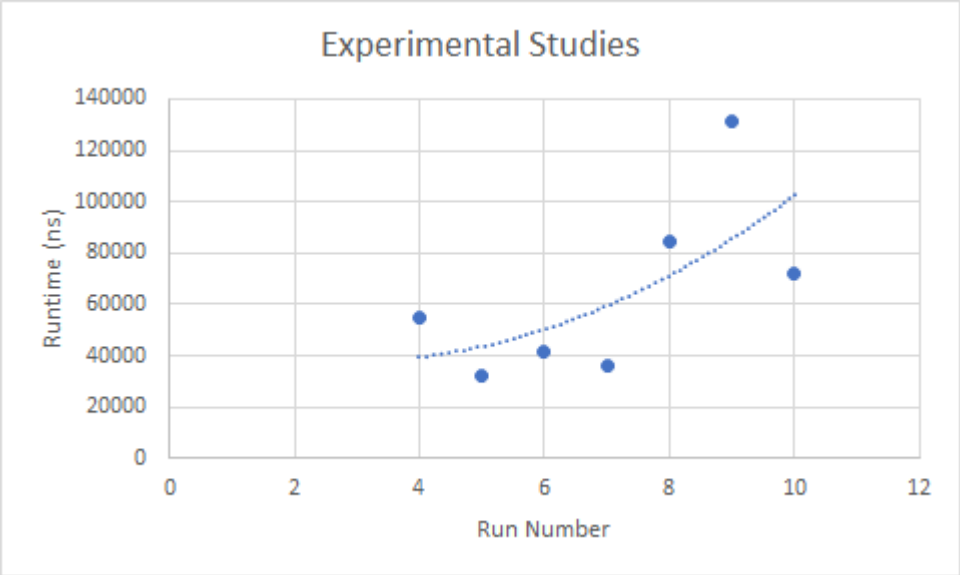

Figure 5 shows the runtimes of the program in nanoseconds with an increasing number of calculations.

With each run, we added one number into the calculation, so that run 4 had to calculate four large numbers, 5 had to calculate five large numbers, and etc. From the table we can see that the runtime is somewhat sporadic but not as spread out as one might think. For example, we are calculating runtimes in nanoseconds, which are $1e10^9$ seconds. Hence, even our largest runtime on run nine took 0.00014 of a second. Not only is this fast, this is our worst case runtime when calculating large numbers. The first three runtimes have been omitted because they were hugely outliers. We can conclude that the first couple runs when the application is first getting into memory is when the runtime takes the longest. Also, the outliers may result from the amount of processes being executed in the CPU and the amount of RAM being used at the time. While this cannot be completely controlled, we have to accept that the environment cannot be shut down completely to run a process. Therefore, since most of the time our program runs in less than 100,000 nanoseconds and our time complexity is O(n^2), we can conclude that our program is fairly efficient and runs quickly. We are impressed with the runtime and even more impressed that our program is able to do everything that we wanted it to do. We have a good base so that we are a able to build on this program and extend its capabilities quite a bit. Using the right packages and conducting more research could mean that our program can be extended to do operations such as solve sets of linear equations, simple calculus, and even plotting functions. We can also conclude that the runtime should have fluctuating runtimes between 20,000 and 150,000 nanoseconds up until there is an overflow of values. Ultimately, we can conclude that our algorithm design could improve but is about as close to being optimized without losing all of the functionalities that it provides.

Task Allocation

Our group project began on March 12th starting with a discussion to determine what our project would be on and after seven days of collaborating, we figured out what we wanted. The program we were going to create was based on students needing help solving equality equations. On March 19th we had begun the programming on our project and each group member was dedicated to a specific piece of code. Lucas Diffey was in charge of creating code to solve for "x" with whatever the user were to input. Thomas Hood worked on getting the GUI to work with Lucas' code. Joseph Gallagher then implemented both Lucas' and Tom's code to communicate with each other to keep track of each problem inputted. On April 1st, each group member was given an assigned task to start coding and in 8 days we would meet to combine everything. Like all projects, we had to make adjustments to ensure each person's code fit with the next person's and on April 12th, we started producing the GUI interface for the user to interact and type out their math problem. After a few days we got the code functioning, and by April 16th, we produced a question for our program, and the program did its job and provided us with an answer giving specific directions on how to solve the problem at hand. Two days later, on April 18th, the GUI was designed and thus our program was completed. On April 20th, as a group, we created a google doc and started allocating specific presentation slides for each member to complete. Three days later on April 23rd, our project and presentation was completed and ready to present. Finally, on April 30th, we decided to make a date for all of us to meet and knock out the paper on one computer so that we were all in agreement of how it sounded.

Conclusion

In conclusion, we were able to produce a very interactive and helpful GUI program which was implemented in C# that enables students both in math or computer courses to get the help they need. Using C# played a huge role in our program since it allowed us to create a professional looking GUI for users to interact with. Also, the implementation of stacks and lists allowed us to generate and equip the correct functionality to solve the complex equations that users provide. The future outcome of our program could go in multiple directions, such as transforming it to handle graphing equations or converting it into an Android/ iOS application. Creating our program allowed us to construct the foundation of never-ending possibilities to solve an infinite amount of problems in the world.

References

Business Insider. *PhotoMath*. Apr, 2018. Digital.

Java Class Library (JCL). *Stack*. Apr, 2018. Digital.

Judith Bishop, R. Nigel Horspool, and Basil Worrall. 2002. Experience with integrating Java with new technologies: C#, XML and web services. In Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande (JGI '02). ACM, New York, NY, USA, 225-225. DOI=http://dx.doi.org/10.1145/583810.583837

Microsoft. (n.d.). ArrayList Class. Retrieved from https://msdn.microsoft.com/en-us/library/system.collections.arraylist(v=vs.110).aspx

Microsoft. (n.d.). List Class. Retrieved from https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx

Miranda Parker and Colleen Lewis. 2013. Why is big-O analysis hard?. In Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling '13). ACM, New York, NY, USA, 201-202. DOI=http://dx.doi.org/10.1145/2526968.2526996

Stephen Wolfram. 2013. Computer algebra: a 32-year update. In Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation (ISSAC '13). ACM, New York, NY, USA, 7-8. http://dx.doi.org/10.1145/2465506.2465930

Zoltan Somogyi and Peter J. Stuckey. 2002. Using the heap to eliminate stack accesses. In Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming (PPDP '02). ACM, New York, NY, USA, 121-132. http://dx.doi.org/10.1145/571157.571170

Appendix

```
public static int Calculate_Expression(String expression)
        {
            // try push and pop on the stack
            try
            {
                // stacks
                Stack<int> numbers = new Stack<int>();
// stack for numbers
                Stack<char> operators = new Stack<char>();
// stack for operators

                // convert the string to a char array
                char[] expression_string = expression.ToCharArray();

                // for loop to iterate
                for (int i = 0; i < expression_string.Length; i++)
                {
                    // skip over blank spaces
                    if (expression_string[i].Equals(' '))
                    {
                        continue;
                    }

                    // if the character is a number
                    if (expression_string[i] >= '0' && expression_string[i] <= '9')
                    {
                        // string builder object
                        StringBuilder build = new StringBuilder();
                        while (i < expression_string.Length && expression_string[i] >=
'0' && expression_string[i] <= '9')
                        {
                            // append the character to the string
                            build.Append(expression_string[i++]);
                        }
                        // push the number onto the stack
                        numbers.Push(int.Parse(build.ToString()));
                    }

                    // if the character is a (, push onto operator stack
                    else if (expression_string[i].Equals('('))
                    {
                        operators.Push(expression_string[i]);
                    }

                    // if the character is a ), push onto operator stack
                    else if (expression_string[i].Equals(')'))
                    {
                        // make sure the top of the operator stack has a (
                        while (!operators.Peek().Equals('('))
                        {
                            numbers.Push(Operation(operators.Pop(), numbers.Pop(),
numbers.Pop()));
// push onto the numbers stack
                        }
```

```
                        // pop the operator stack
                        operators.Pop();
                    }

                    // if the expression string is an operator +,-,*,/
                    else if (expression_string[i].Equals('+') ||
expression_string[i].Equals('-') || expression_string[i].Equals('*') ||
expression_string[i].Equals('/'))
                    {
                        // make sure operator stack is filled and imply order of
operations
                        while (operators.Count != 0 &&
OrderOfOperations(expression_string[i], operators.Peek()))
                        {
                            numbers.Push(Operation(operators.Pop(), numbers.Pop(),
numbers.Pop()));
// push onto the number stack
                        }

                        // push the operator onto the operator stack
                        operators.Push(expression_string[i]);
                    }

                    // if the character is an 'x'
                    else if (expression_string[i].Equals('x'))
                    {
                        // set variable to the x
                        variable = expression_string[i];

                        // make sure there is no out of bounds access
                        if (i > 2)
                        {
                            // if the previous character is an operator (+,-*,/)
                            if (expression_string[i - 2].Equals('*') ||
expression_string[i - 2].Equals('/') || expression_string[i - 2].Equals('+') ||
expression_string[i - 2].Equals('-'))
                            {
                                expression_string[i] = ' ';
// set 'x' to the empty char
                                op = expression_string[i - 2];
// save the operator
                                expression_string[i - 2] = ' ';
// set the operator to the empty char

                            }
                        }

                        // make sure there is no out of bounds access
                        else if (i < expression_string.Length - 2)
                        {
                            // if the next character is an operator (+,-,*,/)
                            if (expression_string[i + 2].Equals('*') ||
expression_string[i + 2].Equals('/') || expression_string[i + 2].Equals('+') ||
expression_string[i + 2].Equals('-'))
                            {
                                expression_string[i] = ' ';
// set 'x' to the empty char
```

```
                                op = expression_string[i + 2];
// save the operator
                                expression_string[i + 2] = ' ';
// set the operator to the empty char

                            }
                        }
                    }
                }

                // whille the count isnt 0 and there are atleast 2 numbers in the number
stack
                while (operators.Count != 0 && numbers.Count >= 2)
                {
                    numbers.Push(Operation(operators.Pop(), numbers.Pop(),
numbers.Pop()));
// push onto number stack
                }


                // return the pop/top of numbers
                return numbers.Pop();
            }
            // catch exceptions
            catch(Exception e)
            {
                // message box for formatting error
                if(MessageBox.Show("Formatting Error", "Error", MessageBoxButtons.OK,
MessageBoxIcon.Warning) == DialogResult.OK)
                {

                }
            }

            // return
            return 0;
        }

public static Boolean OrderOfOperations(char operation1, char operation2)
        {
            // if a parentheses, return false
            if(operation2.Equals('(') || operation2.Equals(')'))
            {
                return false;
            }
            // if operartion is +,-,*,/, return flase
            else if((operation1.Equals('*') || operation1.Equals('/')) &&
(operation2.Equals('+') || operation2.Equals('-')))
            {
                return false;
            }
            // otherwise, return true
            else
            {
                return true;
            }
        }
```

```
public static int Operation(char operation, int number1, int number2)
        {
            // if the operation is *
            if (operation.Equals('*'))
            {
                // add to the solution string
                solution = "Multiply " + number2.ToString() + " and " +
number1.ToString() + " = " + (number2 * number1).ToString();
                how_to.Add(solution);
// add to the overall solution
                return number2 * number1;
// return the product
            }

            // if the operation is /
            if (operation.Equals('/'))
            {
                // if val2 is not 0
                if (number1 != 0)
                {
                    // add to the solutio string
                    solution = "Divide " + number2.ToString() + " and " +
number1.ToString() + " = " + (number2 / number1).ToString();
                    how_to.Add(solution);
// add to the overall solution
                    return number2 / number1;
// return the quotient
                }
                // if val to is 0 show error
                else
                {
                    MessageBox.Show("Cannot divide by 0");
                }
            }

            // if the operation is +
            if (operation.Equals('+'))
            {
                // add to the overall solution
                solution = "Add " + number2.ToString() + " and " + number1.ToString() + "
= " + (number2 + number1).ToString();
                how_to.Add(solution);
// add to overall solution
                return number2 + number1;
// return the sum
            }

            // if the operation is -
            if (operation.Equals('-'))
            {
                // add to the overall solution
                solution = "Subtract " + number2.ToString() + " and " +
number1.ToString() + " = " + (number2 - number1).ToString();
                how_to.Add(solution);
// add to the overall solution
                return number2 - number1;
// return the difference
            }
```

```
    // return
    return 0;
}
```