Iva Tomevska
Magdalena Gunkova

A short write-up about the algorithmic/design choices

Our program starts with getting the input from the text file, which is indicated in the terminal, into a vector. We decided to use vectors because it is convenient as it is dynamically resizing and can expands its memory at run time. It uses a while-loop which runs until there are lines in the text file. It reads line by line, calls a function on every line to separate the variable which needs to be evaluated from the expression (equation) and stores them as two separate elements in the vector.

> The function *devideinput()* which divides the variable from the expression works with searching for the '=' end divides two substrings from the left and right side of the '='. It returns a vector of two elements: variable and expression. Works in O(n) time, where n is he number of characters in the line.

After storing the variables in a vector, we created a map of variable names (string) and variable value (float) pairs. The variable name is the key. We initialized the variables to a default value infinity. We decided infinity because later on, we need to check if every variable is mapped to a proper value, and we can check if there is no value that's equal to infinity (0 wouldn't be a good default value because a variable may be 0). This mapping uses a for- loop that goes through every second element of the vector because it just needs the variables and not the expressions, and it works in $O(\frac{n}{2})=O(n)$, where n is number of elements of the vector.

After we have the map and the vector, we use the vector to search for the variables in the map. If the variable is infinity, we take the expression that is after the variable and evaluate it. For evaluation first we convert the expression to postfix expression and then we solve the equation returning back the answer, which is the value of the variable. If the value of the variable is infinity (which will be if there is another variable in the expression), the loop will continue evaluating until no variable is infinity.

The *inifix2postfix()* function, which converts the expression to postfix has two parts. In the first part it converts the unary operators like '**', '++', '--', or the '-' operator when it's not subtraction, if there are any in the expression, to binary.

In the second part it check if the character is operator or operand and it sorts them in order so that the *evaluate()* function can evaluate them properly.

It uses a stack when it converts and sorts everything and it returns a string of the characters in the stack. Because it loops twice trough the string expression (once to check and convert the unary to binary operands, and once to sort the operators and operand in order) it works in O(2n)=O(n) where n is the number of characters in the string expression.

The *evaluate()* function goes through the postfix string, checks for if an operand is a variable or a number. It uses a stack to evaluate the expression. If it is a variable it returns infinity if the variable is not yet evaluated. If it is evaluated, it pushes the mapped value of the variable to a stack and treats it like a number. If it is a number it pushes the number to the stack. When it encounter and operator, it takes the first two values of the stack, evaluates them and pushes the evaluated value back to the stack. It returns a numeric value of the evaluated expression or infinity if there's unevaluated variable. It loops as long as there are characters in the sorted string, so it runs in O(n) time , where n is the number of characters in the string.

The overall worst-case time complexity of our algorithm is O(n^2) because in our main function there is a nested for-loop in the main while loop, which goes through the vector where we store the variables and the expressions- O(n) time complexity, and the while loop runs while there are no infinity values in the vector- O(n) time complexity.