1. Approximately 4 hours total
2. Since this was an individual project, I did not need to use github or any sort of code sharing mechanism. I made sure to separate everything into functions so that my code was neatly organized and could be reused efficiently.
3. Python, invoked the sys library for reading input.
4. The format of the input is the same as specified in the test files. Each expression is on a separate line, with the output following a line of '-'. If input is from the keyboard, a new expression should be inputted on each line. For output, each step of the reduction is shown, with each line of the reduction beginning with a '=>'. This continues until the final reduction is outputted, and then the program starts the next expression in the file, or waits for the next expression to be typed in. Only the final reduction is shown for arithmetic expressions.
5. Combinators: S, K, I, B, Y, W, C, P, T, J
   Built-ins: +, -, *. / (can only process single digit numbers)
6. My program continually converts the expression between a list and a string. A list where each argument is an entry makes processing each argument for the combinators very efficient. Any time I encounter a '(', other than at the very beginning of the expression, I set a pointer to that location and keep going across the string until the matching ')' is found. Everything between these parentheses is an expression that needs to be grouped together and is inserted as a single argument in my list. After processing a reduction with a combinator, I convert my list back to a string (adding back the necessary parentheses) so that my list can be reformatted correctly. For example, in K(SI)abc, the first reduction leaves (SI)bc. Originally, (SI) was treated as one argument, but now that it leftmost, I need to use the S operator with I as the first argument. Reformatting the list as a string allows me to re-traverse the string, eliminate the leftmost parentheses, and separate as needed. To process the arithmetic, I did not utilize leftmost order, but instead used the typical order as described in the lecture notes. I used a binary tree to keep each layer of parentheses separate so that the arithmetic could be calculated properly.
7. I tested my SKI functions by using the test files provided. For the supercombinators, I created a test case to check the definition of the supercombinator to ensure it works properly. For the arithmetic, I wrote a few expressions for each operator, as well as a few expressions combining operators. To further check my SKI combinators, I created boolean operators NOT, AND, OR, NAND, and XOR. In terms of boolean logic, True = K, and False = SK. These can be used to create boolean functions NOT,AND,OR as follows:
NOT = (F)(T) = (SK)(K)
(T)NOT = (T)(F)(T) = F
(F)NOT = (F)(F)(T) = T

AND = F = SK
(T)(T)AND = (T)(T)(F) = T
(T)(F)AND = (T)(F)(F) = F
(F)(T)AND = (F)(T)(F) = F
(F)(F)AND = (F)(F)(F) = F

OR = T = K
(T)OR(T) = (T)(T)(T) = T
(T)OR(F) = (T)(T)(F) = T
(F)OR(T) = (F)(T)(T) = T
(F)OR(F) = (F)(T)(F) = F

source: https://en.wikipedia.org/wiki/SKI_combinator_calculus

NAND and XOR can subsequently be created as follows:
NAND = (AND)(NOT) = (F)(F)(T) = (SK)(SK)(T)
(T)(T)NAND = (T)(T)(F)(F)(T) = F
(T)(F)NAND = (T)(F)(F)(F)(T) = T
(F)(T)NAND = (F)(T)(F)(F)(T) = T
(T)(F)NAND = (F)(F)(F)(F)(T) = T

XOR is constructed by ANDing the NAND and OR
(T)XOR(T) = ((T)(T)NAND)((T)OR(T))AND
= ((T)(T)(F)(F)(T))((T)(T)(T))(F) = F
(T)XOR(F) = ((T)(F)NAND)((T)OR(F))AND
= ((T)(F)(F)(F)(T))((T)(T)(F))(F) = T
(F)XOR(T) = ((F)(T)NAND)((F)OR(T))AND
= ((F)(T)(F)(F)(T))((F)(T)(T))(F) = T
(F)XOR(F) = ((F)(F)NAND)((F)OR(F))AND
= ((F)(F)(F)(F)(T))((F)(T)(F))(F) = F

From my output you can see that all of the boolean functions are calculated correctly using only S and K.