1. For simple regular expressions - be able to produce the state diagram that matches with the regular expression and identify what strings will be matched and if a string will be rejected by the regular expression. Know that regular expressions are equivalent to some deterministic or non-deterministic automata. Know that you can't use a regular expression to find matching (balanced) parenthesis.

Examples:

```
^a$                 Matches the letter a on a line by itslf
^ab*$               Matches the letter a followed by any number of
                    b's on a line
^abc                Matches a line that starts with abc
abc$                Matches a line that ends with abc
[0-7]               Matches an octal digit 0, 1, 2, 3, 4, 5, 6, 7
[a-f]               Matches a lower case a, b, c, d, e, f
[0-9a-fa-F]         Matches a hex digits both upper and lower case
[0-9][0-9]*         Matches 1 or more decimal digits
[-+][0-9][0-9]*     Matches a plus or minus followed by a decimal number
                    atleast 1 long
```

Be able to draw out a Non-Deterministic Finite State Automata (NDFA) for these regular expressions. Use "BOL" for beginning of line, `^` and EOL for end of line `$`.

Use 'a-f' or similar to for matching a range of characters in your NDFA.

2. List the different kinds of machines, Logic, Finite Automata, Push Down Automata, Turning Complete.

3. Know that a regular expression (or the corresponding Finite Automata (DFA, NDFA)) can not match parenthesis or parse HTML because it requires matching and counting. To do these tasks requires a Push Down Acceptor.

4. Know that you can not write a program that will test other programs and find out if they complete.

5. Know how to take a set of boolean logic and generate the corresponding and/or/not gates in a diagram.

6. Know how to take a simple diagram with some and/or/not (or nand/nor/not) gates and extract the boolean logic from it.

7. Be able to read a truth table and derive the set of equations from it. Read a truth table and determine what output are turned "on" for a given input.

8. Convert decimal to hex and hex to decimal. Know that 0b001 is in binary and that 0x0A is in hex.
9. Know that Unix/Linux uses `\n` for an end of line and that Microsoft Windows uses both `\n` and `\r` for an end of line.
10. Know how to use an ASCII table to look up characters.
11. Know that there are more characters than just ASCII - be aware of Unicode.
12. Be able to look through a simple program in MARIA (our assembly language) and figure out what the program will do.
13. Be able to look at the output of an `od –c` (octal dump) and identify “bad stuff” - characters that are hidden - or that it is not in the correct format for the system that you are on.
14. Be able to take the 1s and 2s compliment of a number.

15. For a simple MARIA program be able to trace through what the program will do. For example, what is the expected output for:

```
     Load X          // Tmp = X
     Store Tmp
     Load Y          // t2 = Y
     Store t2

L0,  Load Tmp        // Tmp = Tmp - 1
     Subt _1
     Store Tmp
     SkipGt0         // Skipcond 400 - If AC <= 0 goto  L1
     Jump L1
     Load t2         // t2 = t2 + Y
     Add Y
     Store t2
     Jump L0         // Goto Top Of Loop L0
L1,  Load t2         // Output t2
     Output
     Halt             // End of Program Running


X,     DEC    2
Y,     DEC    8
_1,    DEC    1
Tmp,   DEC    20
t2,    DEC    0
```

The output that I get from running the emulator is:

```
PC[0000/0x0000] AC[0000/0000] Mem[]=0x1010 Instruction: 0x1000 | Load      Operhand: 0
PC[0001/0x0001] AC[0002/0002] Mem[]=0x2013 Instruction: 0x2000 | Store     Operhand: 0
PC[0002/0x0002] AC[0002/0002] Mem[]=0x1011 Instruction: 0x1000 | Load      Operhand: 0
PC[0003/0x0003] AC[0008/0008] Mem[]=0x2014 Instruction: 0x2000 | Store     Operhand: 0
PC[0004/0x0004] AC[0008/0008] Mem[]=0x1013 Instruction: 0x1000 | Load      Operhand: 0
PC[0005/0x0005] AC[0002/0002] Mem[]=0x4012 Instruction: 0x4000 | Subt      Operhand: 0
PC[0006/0x0006] AC[0001/0001] Mem[]=0x2013 Instruction: 0x2000 | Store     Operhand: 0
PC[0007/0x0007] AC[0001/0001] Mem[]=0x8800 Instruction: 0x8800 | SkipGt0   Operhand: 8
PC[0009/0x0009] AC[0001/0001] Mem[]=0x1014 Instruction: 0x1000 | Load      Operhand: 0
PC[0010/0x000a] AC[0008/0008] Mem[]=0x3011 Instruction: 0x3000 | Add       Operhand: 0
PC[0011/0x000b] AC[0016/0010] Mem[]=0x2014 Instruction: 0x2000 | Store     Operhand: 0
PC[0012/0x000c] AC[0016/0010] Mem[]=0x9004 Instruction: 0x9000 | Jump      Operhand: 0
PC[0004/0x0004] AC[0016/0010] Mem[]=0x1013 Instruction: 0x1000 | Load      Operhand: 0
PC[0005/0x0005] AC[0001/0001] Mem[]=0x4012 Instruction: 0x4000 | Subt      Operhand: 0
PC[0006/0x0006] AC[0000/0000] Mem[]=0x2013 Instruction: 0x2000 | Store     Operhand: 0
PC[0007/0x0007] AC[0000/0000] Mem[]=0x8800 Instruction: 0x8800 | SkipGt0   Operhand: 8
PC[0008/0x0008] AC[0000/0000] Mem[]=0x900d Instruction: 0x9000 | Jump      Operhand: 0
PC[0013/0x000d] AC[0000/0000] Mem[]=0x1014 Instruction: 0x1000 | Load      Operhand: 0
PC[0014/0x000e] AC[0016/0010] Mem[]=0x6000 Instruction: 0x6000 | Output    Operhand: 0
Output: 16/0x10 Char -><-
PC[0015/0x000f] AC[0016/0010] Mem[]=0x7000 Instruction: 0x7000 | Halt      Operhand: 0
```