1. a) A '-' in the original transition table implies that we don't care what the input is, whether the input is '0' or '1', we do the same thing. Therefore, we can replace the original line containing a '-' with 2 lines having identical values in all fields, except we'll replace the '-' by a '0' in the first line and a '1' in the second.

   example: \[ \begin{array}{c|c|c|c|c|c} \hline \text{lost} & 1 & 1 & \text{rccw} & 0 & 0 \\hline \end{array} \]

   becomes

   \[ \begin{array}{c|c|c|c|c|c} \hline \text{lost} & 1 & 0 & \text{rccw} & 0 & 0 \\hline \end{array} \]

   \[ \begin{array}{c|c|c|c|c|c} \hline \text{lost} & 1 & 1 & \text{rccw} & 0 & 0 \\hline \end{array} \]

b) use equation: \[ \# \text{ of states } = 2^{(0+5)2^{1+3}} \]

   where \[ 0 = \text{output} = 3 \]
   \[ i = \text{inputs} = 2 \]
   \[ s = \# \text{ of bits to encode a state} = 2 \]

   then, \[ \# \text{ of states } = 2^{(3+2)2^{1+2}} = 2^{5+4} = 2^{80} \]

c) TL + TR + F columns determine the operations the ant can perform.

   'Next' column determines order of operations.

d) They cannot be uniquely encoded.

2. a) Negating a number $a$ is equivalent to the expression $a' = \overline{a}$
   therefore we can use $\text{sub } a, 0, a$

b) For this operation we need to use a logic gate which will give us an output 1 for input 0, and an output 0 for input 1.

   XOR gate can do that provided the 2nd input is always 1

   XOR: $a', a, 0$ OR FFFF
c) the shift right operation accomplishes this task if we shift the leftest bit 15 times
\[
\text{SRA} \hspace{1cm} d, \hspace{1cm} a, \hspace{1cm} 15
\]

d) clearing a register to a value 0 is the same as setting it to the sum of 0 and 0
\[
\text{add} \hspace{1cm} d, \hspace{1cm} 0, \hspace{1cm} 0
\]

e) this is just a branch if equal instruction that's always true/equal
\[
\text{beq} \hspace{1cm} d, \hspace{1cm} 0, \hspace{1cm} 0, \hspace{1cm} \text{label}
\]

3) simply put these instructions into the assembly simulator, and check the contents of the 'contents' field next to the current 'instruction'
\[
\begin{array}{c|c|c}
\text{operation} & \text{hexadecimal} & \text{binary} \\
\hline
\text{add} \hspace{1cm} d, \hspace{1cm} a, \hspace{1cm} b & 0x0000 & 0000 0000 0000 0000 \\
\text{shl} \hspace{1cm} d, \hspace{1cm} 10, \hspace{1cm} 4 & 0x9a4 & 1001 1010 1010 0100 \\
\text{xor} \hspace{1cm} d, \hspace{1cm} 11, \hspace{1cm} 12 & 0x3abc & 0011 1100 1011 1100 \\
\text{sub} \hspace{1cm} d, \hspace{1cm} 1, \hspace{1cm} 1 & 0xe14 & 1110 0101 1001 0001 \\
\text{beq} \hspace{1cm} d, \hspace{1cm} 0, \hspace{1cm} 0, \hspace{1cm} 0, \hspace{1cm} \text{label} & 0x0000 & 0000 0000 0000 0000 \\
\end{array}
\]

4) the idea here is to contrast the length of the code with how long it takes for that code to execute and arrive at the solution (ie instruction count)
- MultA : 9 lines of code, 98 steps
- MultB : 12 lines of code, 35 steps

b) MultA does not support the first argument being negative, because it is designed to add the 2nd operand (1st operand) number of times, which doesn't make sense (can't add a number a neg. number of times).
- MultB works in all scenarios.
- Fix for MultA : store the sign bit of first operand somewhere, if it's a neg. number convert it to positive, then multiply, change the sign of result if first operand was negative.
# find the maximum value in an array
main:    addi $1, $0, array
         addi $2, $0, 10
         beq $15, $0, $0, max
halt:   beq $0, $0, $0, halt

array:  .data 634, 32, 460, 902, 343, 956, 28, 587, 460, 202

# your code goes here
max:    add $3, $0, $0           # set max to 0
loop:   beq $0, $2, $0, done    # if length is equal to 0, we're done
        subi $2, $2, 1          # decrement our length by $1
            ld $4, $1           # load new number into $4
        addi $1, $1, 1         # update our $1 to point to next number for future
        sgt $5, $4, $3         # compare current number ($4) to the current max ($3)
            beq $0, $5, $0, loop # if $5 is = 0, restart loop
            add $3, $4, $0       # otherwise, put current number into max
        beq $15, $0, $0, loop   # restart loop again

done: