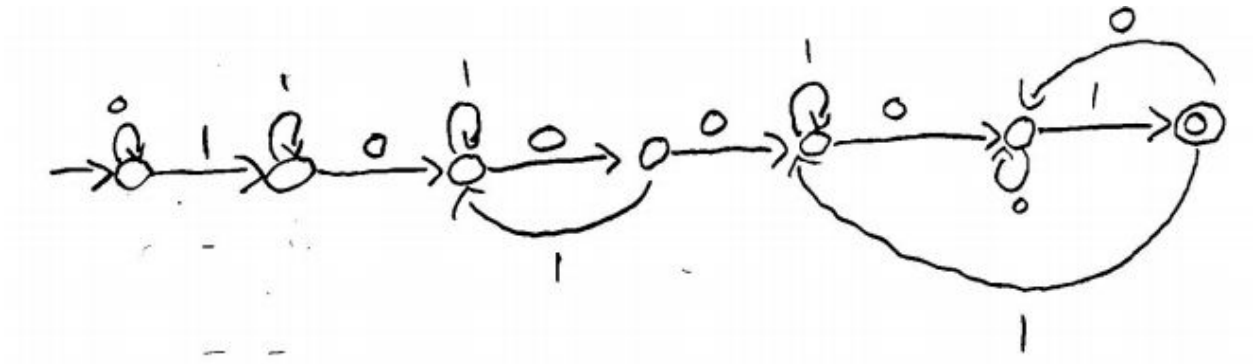
Dennis George
CS360 Test 1
Start Time:10:30
End Time: 12:30
Attempted problems: 1,2,3,4,5

1.



This DFA will work because the only way to transition to the next state is if you meet the requirements of the state before. The problem requires the binary string to begin with a 10 combination. The only way to transition out of the first state is by encountering a 1. If another 1 is encountered, then we remain in this state. We move forward to the next state only when a 0 is required, which guarantees that we meet the first requirement of the problem - binary string must start with 10. The next requirement is that the binary string must contain 00. Now that we are at a state where we have guaranteed to pass the first requirement, the DFA needs to be able to handle any amount of 0's and 1's and should only continue once we have encountered 00. At any point we find a 0, we move to the next state. If the immediate next symbol is a 1, we go back to the previous state. We only move forward when we encounter two 0's back to back. Again, we don't care how many 1's and 0's occur at this point. If a 1 is encountered we remain at the same state. The final condition of the problem is that the DFA must end in 01. When we encounter a 0, the next symbol has to be a 1 and at which point we can terminate. If we encounter another 0, then we are still in the same state. Only when we encounter a 1 after a 0 will we move to the final accepting state.
This DFA is minimal because each state is reachable and distinguishable. There are no unreachable states to be eliminated and there are no non distinguishable states that can be merged.

2.
S -> 00S | 11S | 01T | 10T
T -> 00T | 11T | 01S | 10S | ε

Odd number of 0's and 1's -> There can be any number of double 0s (00) and double 1's (11) in a row. Every time we encounter an odd combo (10 or 01), we should be able to terminate the string which is shown in T. Once in state T, adding any combination of double 0s or double 1s will keep us in a state where we should be able to terminate and accept the string. If we encounter another odd combo, we are back to the original state and can no longer accept.
Justification for non ambiguous -> We are always parsing left to right so the branching will always go towards the right

3. (i) pre-order

```
(define (pre-order tree)
  (if (null? tree)
      '()
      (append (cons (weight-leaf tree)
              (pre-order (left-branch tree))
              (pre-order (right-branch tree))))))
```

(ii) in-order

```
(define (in-order tree)
  (if (null? tree)
      '()
      (append  (in-order (left-branch tree))
               (cons (weight-leaf tree)
               (in-order (right-branch tree))))))
```

(ii) post-order

```
(define (post-order tree)
  (if (null? tree)
      '()
      (append  (post-order (left-branch tree))
               (post-order (right-branch tree))
               (cons (weight-leaf tree)))))
```

These implementations are based off of the normal tree traversal shown in SICP 2.63. Pre-order traversal requires you to first visit the current node you are on, second traverse the left subtree, and finally traverse the right subtree. Whenever we reach a node, we want to append the weight of that node to our returned list that will have all of the weights. For in-order traversal, the idea is the same except we need to traverse the left subtree first, then visit the current node, then traverse the right subtree. Post-order will visit the left subtree, then the right subtree, and then visit the current node. These scheme implementations are correct because they will successfully traverse the trees in the proper order. Since all implementations are recursive, the function will traverse the entire tree, adding weights to the returned list in the proper order.

4. The condition `(if (leaf? next-branch)` is responsible for determining when the decoder is ready to evaluate the current branch to its appropriate symbol. If we did not have this condition, then the algorithm that we used to base our Huffman tree off of would have lost its integrity. The decoding of the tree operates under the assumption that once you hit a leaf node, we have to evaluate. If the decoder does not evaluate when a leaf is found, then it may continue to search for leafs, and could decode our bits to an improper symbol. This conditional is responsible for determining if the decoder has found a proper symbol and recursively decoding with the entire tree. If the decoder has not yet found a symbol, it will continue to decode recursively with the appropriate next branch.
The depth of `(decode-1 bits tree)` depends on the length of the string we are decoding and the height of the tree. In a balanced tree, it will be called log2(n) times as we need to traverse the entire height of the tree in order to find a leaf node. If the tree is not balanced, then there will be worst case “n” recursive calls until we find the leaf node that we are looking for in order to find the proper symbol.

5.
Target of computation: This algorithm is meant to simulate the execution of a DFA. The simulation will either accept or reject the input. The DFA is defined as a list containing three items: the start state, the transition function, and a list of final states. Given this description and input symbols, the (move) function searches for a transition from the start state to a new state “s”. The (move) function will then return a new machine that has “s” as its start state. (simulate) has a tail recursive (helper) function that accumulates the list of moves and will return when all of the input symbols have been consumed. The wrapper function of (simulate) will determine if the (helper) recursive function has ended in a final state, making the DFA either reject or accept the input symbols.

Data Structures Used: Lists are used to represent a DFA

Language Mechanisms:
Selection: Throughout the algorithm there are plenty of places where we execute different instructions based on the response from a conditional operation.
Procedural abstraction: The programmer now has the ability to use this function to simulate any DFA they have constructed. The simulate algorithm does not depend specifically on one particular DFA and can work for any proper DFA that is passed in.
Recursion: The simulate algorithm depends on recursion in order to consume all of the input symbols.
Exception handling and speculation: In the (move) function, we have a check for if the current state that we just calculated has returned an error. The (helper) function also has base cases in order to prevent a stack overflow from occurring.