Name: *Quinten Lambert John Hellrung*

This reflection is to be completed individually, though consultations with TAs and classmates are encouraged as long as they are appropriately acknowledged.

This assignment was intended for you to implement the binary full adder that you and your team designed for the team activity. You were asked to consider how to handle the bit "carry" that can result from adding two bits and how it can "ripple" up through the addition.

This reflection document is intended to help you think about how to make the leap from design implementation and the possible challenges/changes you may need to make in the process. Copy the pseudocode design you created with your team for adding two BinaryNumber objects, i.e. "C = A + B", assuming that A and B are BinaryNumber objects:

|  |
| --- |
| Def add (self, B1, B2)  Have each binary number get a value that is set to their least significant bit.  Set a value called carry equal to 0 indicating that there is no carry  Check that the bit value in each list is not None, if so go into a loop where values are checked depending on the bit that you are on. If those values are found to equal certain values then set the next bit coming up to a certain value and put it into another linked list that will hold the final added value, then iterate through each bit in the binary number, repeating the process. |

In the team exercise, you also encountered questions about boundary conditions. What conditions did your team find and how did it impact the way that you implemented the addition operator?

|  |
| --- |
| The condition when say a 4 bit binary number is added with another 4 bit binary number and the result would be that of a 5 bit binary number. This would cause problems if one would not plan for this and add another bit at the beginning of the number. Therefore the implementers must add a way to add that extra bit when the binary numbers equal a bit larger than the bit value that they are already. |

Where there any assumptions built into the design that you were not aware of until you actually tried to implement this function? What were they, and how would you expose these assumptions when designing programs in the future?

|  |
| --- |
| My partner and I assumed that the binary numbers would add another bit when necessary as they were added together and until we tried to implement the code, we were completely unaware of it. This caught us off guard and caused us to work a lot more on the code than we initially anticipated. In order to expose these kind of assumptions right off the bat, one should attempt to create the program as early as possible before the due date to realize all possible problems as they can at that given time. This would allow a sufficient amount of time to bring the code together and fix all of the issues (or most of them) before turning in the final draft of the code. |

What details were missing from the design that you needed to add when implementing addition? How do you think you can make sure to consider every detail in future designs?

|  |
| --- |
| Some of the details that were missing include, checking for overflow handling when the bits didn’t quite add up. Another detail that may have been missing would have been the fact that if the binary numbers were of different bit lengths. In order to fix this one would have to take the shorter bit length and make sure it added up to the be the same as the larger bit length by adding 0’s to the beginning of the number as placeholders. To be sure to consider every detail in future designs one must first write out every possible combination of what the code is trying to implement before actually coding it and then making sure that the code covers each of these outcomes. |

Make the modifications to the design given your experience in implementing this method and write the updated design pseudocode below:

|  |
| --- |
| def add(self, binary\_1):  """Adds the binary1 and binary2 and sets the answer to problem as self  Post: print the binary of the add two binary numbers"""  #-------------------------------------------------------------------------------  def TorF\_Conv(value):  """Sets the value of True or False to an int of 1 or 0  Post”: Returns 1 if input is True else returns 0 """    def length(bit):  """finds the lenght of the bit numbers  Post: Returns with number bits that are in a BIt"""  def reString(List):  """Rearainages the list  Post: returns list named String”””  #-------------------------------------------------------------------------------  set b1 to the leastSignificantBit or bit\_ref zero  set b1 to the leastSignificantBit or bit\_ref zero  VL = length(b1) #finds the length of the linked list "b1"  VL2 = length(b2) #finds the length of the linked list "b2"  Make one logic statement saying that if the length for b1 and b2 are the same then set b1 to A and b2 to B  Make another logic statement saying that if the length of B1 is less than B2 then ass zeros to B1 to make B1 the same length  Make another logic statement saying that if the length of B2 is less than B1 then ass zeros to B2 to make B2 the same length  makes a list  makes a list  sets a variable to zero  create a Loop until the node equal None  Gets the value of the bit note for A  Gets the value of the bit note for B  convers boolens to int the function TorF\_Conv for A  convers boolens to int the funcaton TorF\_Conv for B  Adds the value of the varable "value" and value of the varable "value2" and store the result as "new\_value"  Append new\_value's value to a list  Run logic that says that if new\_value equal 2 and carry equal 1 then append 1 to the list named "List", else append 0 to the list    sets the value of "carry" to 1  sets the value of "carry" to 1  Run another logic statement that says that if new\_value equal 1 and carry equal 1 then append 0 to the list named "List", else append 1 to the list  sets the value of "carry" to 0  Runs logic that says that if new\_value equal 0 and carry equal 1 then append 1 to the list named "List", else append 0 to the list  sets the value of "carry" to 0  moves the bit\_ref to the next node so we can see the next value for A  moves the bit\_ref to the next node so we can see the next value for B  Logic statment saying that if the last item in the list is 2 then append 1 to the list called "List"  prints for the user |