A) So for my solution I implemented what the book recommended. Developing a graph where the vertices of the graph are the strings we are looking to super string. The graph is a fully connected graph, and we weight the edges according to a cost to get from the one to another. Once this is set up the solution is to find a minimum spanning tree, and concatenate the results, ignoring overlap, to get our resulting super string. We can repeat this choosing a different starting node to get forest of results and select the shortest of the resulting super strings. This returns our shortest super string.

B) Pseudocode:

Class Node(string, id):  
 \*\*Constructor\*\*  
 self.edges = []

def addEdge(node):  
 substring= “ ”  
 weight=0  
 for char in node.string:  
 substring += char  
 if not substring in self.string:  
 weight = weight+1  
 self.edges.append((node.id, weight))

Class Graph(string \_array[]):  
 \*\*\*Constructor\*\*\*  
 For string in strinArray:  
 addNode(string)

def fullyConnect():

\*\*\*Fully connect the graph\*\*\*

Def superstring(node):

Node.visited = True  
accumulator = “ “  
for char in node.str:  
 accumulator += char  
 if not accumulator in self.superString:  
 self.superString += char //superstring is our result  
sort = sorted(node.edges, sort by weight)  
for node in sort:  
 if not visited:  
 superstring(node) //Depth first spanning tree  
 break

C) We can add a level of memoization to the algorithm by remembering valid prefixes from previous words and testing new words for these prefixes.

D) The algorithm works in a left to right read fashion matching the prefix to the already created super String. These matches ensure we don’t add unnecessary characters. Following this checking each starting node asses which spanning tree yields our smallest possible substring and that is our result.

E)