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) The big -O notation of this problem is easiest to represent by the number of words to compare (S) and the number of letters in each word (L). For building the fully connected graph we need to do a comparison of each node to every other node getting us O(S,L) = S\*(S-1)\*(L)

Or really O(S,L) = L\*(S^2). Once the graph is completed, the solution process involves visiting each node on a path following the lowest cost and comparing it to the current superstring. This gets us O(S,L) = S\*L

The graph completion is the computationally expensive piece of this algorithm but I believe during that process we could also be building our solution. This could be a potential speed up of the current Big-O.

F) With a sparse matrix we could consider every row as an individual string. We then would run the algorithm proposed finding s superstring representation, that could hold the non-zero elements of the matrix. Additionally, we could rework it to hold the number of zeros we minimize on any given row.

R)

This project required a good amount of reading and re-reading. Imagining the entire problem as a graph was not my initial guess by any stretch. After reading about a proposed algorithm and doing a little illustrating a graph solution made clear sense. Then just pick a min-spanning tree and run with it. Another interesting application of graph problem-solving.