5. The CKY algorithm is a dynamic-programming solution to the problems of structural ambiguity that parsing algorithms are faced with. Namely, the two types of structural ambiguities that are detailed in Jurafsky and Martin: attachment and coordination ambiguity. Each of these structural ambiguities results in the same problem for the parsing algorithm-- "when the grammar can assign more than one parse to a sentence". This problem arises from the fact that a context free grammar, true to its name, does not utilize context to specify \textit{how} a certain sentence should produce parse tree, it is rather a "purely declarative formalism", and as such it cannot inform the algorithm which tree would more appropriate when faced with a structural ambiguity. NLP systems must be able to "choose" only one parse for a given sentence, but when there are multiple options due to an ambiguity, there is a need for syntactic disambiguation algorithms, for they can inform the parsing algorithms of elements they would not otherwise have recourse to: namely, statistical, semantic, and contextual information.

A dynamic programming approach is beneficial because it will systematically fill in all the possible parse trees for a given input, and one we have that information, then all the tools necessary to tackle the problem of which to choose will be available. CKY requires that the grammars we use as input be in Chomsky Normal Form (CNF). We need to input rules in our grammar that do not mix terminals with non-terminals. In order to do that it is necessary to "introduce a dummy non-terminal that covers only the original terminal." In order to do that we keep the rules that conform to CNF, and convert the terminals within the rules to dummy non-terminals. The next step in converting the grammar to CNF is to convert unit productions, and finally to make all rules binary before adding them to the new grammar. We can now be sure that every non-terminal node in a parse tree will break off into two separate offspring at the level above the part of speech.

This is useful because we will encode an entire tree using a two dimensional matrix, and each cell in this matrix will have the format [i,j] and will contain the constituents ranging from the index of positions i through j as specified in the input in the form of the non-terminals that represent them. Therefore, for every constituent that [i,j] represents, there will be a corresponding position input \textit{k} that bifurcates into two, which will mean that the first constituent [i,k] will be someplace to the left of entry [i,j], and somewhere on row \textit{i}. More concretely, the algorithm will fill the cells going from bottom to top, and moving from left to right, so that every entry that will contribute to the cell it is currently working on would already be present in the data at that particular point. Columns on the left need to be filled first, and also columns below since the algorithm works from bottom to top. The leftmost loop works on the columns, and the loop within works over the rows. The innermost loop works as k scans through the cells [i,j] moving from left to right through row \textit{i} and bottom to top through row \textit{j } for the possible places where a string may split. Whenever the algorithm finds a place where there is a possible split, it considers the two spans and whether a possible combination of the two is possible within the bounds of the grammar. The non-terminal on the left-hand side of said rule will then be added to the table.

It should be noted that at this stage, the algorithm works as a recognizer, not a parser. In order to make it into a parser, we need to store which existing constituents were combined to make each of the new constituents we input into the table. Since each possible parse is weighed equally at this point, and there could be quite a few possible parses for any given input, we will need to tweak this algorithm to include probabilities or use another fix like partial parsing to bypass this issue.