Dalton Altstaetter

EE 360C

Dr. Pedro Santacruz

11/27/2014

Program 3 Overview

**Part 1 – Pseudocode & Approach:**

How I plan to implement this program is to use dynamic programming (DP) to create the sentences and add the necessary white spaces. However, when doing DP I ran into some problems and tried a brute force method that worked on the first try, so that is what I really implemented and abandoned the idea of DP. I created a recursive function that created the sentences. I did the following:

---> Read the input from the input file to get the information for the number of words in the dictionary and the sentence which needs to have whitespace added.

---> I stored the dictionary words in a dynamic data structure which I will iterate through when searching for words. If duplicates were present in the dictionary input being read in then I only added the first occurrence. Capital and lowercase words are treated as separate words. For example if “The” and “the” were words in the dictionary input I would add both of those words to my dictionary data structure.

---> Next I parsed checked the sentence and only added the dictionary input words to my dictionary data structure if and only if those words were contained in the sentence being parsed. This improves performance, reduces memory use, and simplifies the program

---> Next I call my recursive function that creates all my possible sentences. This function returns all my output sentences that I will output to stdout.

---> My recursive function gets passed the remaining sentence that is left to be parsed for whitespace. Initially the input sentence is passed to this function unaltered. My base case checks to see

If( remainingString.length() ⩵ 0)

If this is true then I have finished creating one of the output sentences. When this occurs I store all the words in an output ArrayList which I used to store all the intermediate words when creating this sentence.

---> When I am not at my base case I iterate through all the words in the dictionary to see

For(int i = 0; i < Dictionary.size(); i++)

{

If(remainingString.startsWith(Dictionary.get(i)))

{

removedPrefix = removeFrontString( remainingString, Dictionary.get(i));

stack.push(Dictionary.get(i) + “ “); // store matching word and add space in sentence

Recurse(removedPrefix); // removedPrefix is the remainingString w/front word removed

stack.pop();

}

}

To explain the above code this shows how I iterate over all words in the dictionary to look for words that the remaining sentence starts with. When it finds a match it adds that word to the stack which contains a valid sentence being constructed and then continues with the remainingString with the matched word on the front removed. To enumerate all the possible sentences I pop that word off the stack when it returns to this point in the code which will check for other possible matches for words that could be matched to the beginning of the remaining string. The “Recurse(String remainingString)“ function is the recursive function I am using to construct the sentence and is where this above code is located in.

At the end I output the number of sentences I made and iterate through the sentences outputting each sentence on a new line.

System.out.println(outputSentences.size());

For(int i = 0; i < outputSentences.size(); i++)

{

System.out.println(outputSentences.get(i));

}

This basically sums up my algorithm. To review I go each word in the dictionary looking for words that match the beginning of remaingString and I remove that matched word from remainingString and push it onto a stack (with whitespace added) which holds the intermediate valid sentence being constructed. When the remainingString.length ⩵ 0 then I read the stack from back to front to recreate the sentence. After doing so I return and pop the last added word off the stack and look for all other possible matches

**Part 2 - Runtime Complexity**:

1. Let’s say there are n words in the dictionary that could match words in the sentence. To read in the words and the sentence is O(n+2) for n words plus the sentence plus the number of words in the dictionary.
2. Next, I look to see if there are any words in the dictionary input that aren’t contained in the sentence. This is O(n).
3. Next, I get into my recursive function. The worst case runtime occurs when there are several words that the front of the remainingString could get matched to. The best example of this is the sample input with sentence “aaaaaa” and dictionary words = {"a","aa","aaa","aaaa","aaaaa"}; in which case the recursive function gets call n! times. During any one of those n! the runtime for a single call on the worst case is O(n) since I would have to look through all dictionary words for a match. If I hit my base case I pop all strings off the stack, reverse them, then add them to an output String, this is O(n+1) to reverse n words and store that result. This makes the worst time run complexity for a single recursive call to be O(n) and since this occurs n! times, This is on the order of O(n\*n!);
4. Following this all I need to do is print out the number of sentences I found and then print out the sentences themselves. Let m be the number of valid sentences I found when doing C. above. This makes this O(m+1) to print m sentences and the number of sentences.

To find the total Complexity of my algorithm Sum up the complexities A. + B. + C. + D. to get O(n+2) + O(n) + O(n\*n!) + O(m+1) where m <n\*n! ⇒ (Assume m = n\*n!) then O(2n + 2\*n\*n!) = O(n\*n!)

**Part 3 –** Prove that your algorithm is guaranteed to provide the correct result given a valid input.

Explain how you validate and verify your algorithm.

I know my algorithm is guaranteed to provide correct results given a valid input because I enumerate all possible values that my sentence could potentially take on. I am using brute force and check every possible word to match the front of the sentence. Its validity and verification can be assumed as long as I have a working algorithm since I am never taking any shortcuts and never optimize anything thus guaranteeing that I’m not missing any possible valid sentences. Therefore if my algorithm provides atleast one valid output sentence I can guarantee it will provide all valid output sentences.