Lab 2 – “Dash Dot Dot Dot Dash” Write-Up

Secret Message:

**1) ....-.-.-.---..--.-..--..-...--..-.........**

**2) -.....-.-.----.........-----.-.-.-...-.--.**

**3) .--..-...-..-..----..**

**4) .--.-...--..-.-.--.-.----.**

1) IF YOU CAN READ THIS

2) THE CODE IS WORKING

3) WELL DONE

4) AND PARTY ON

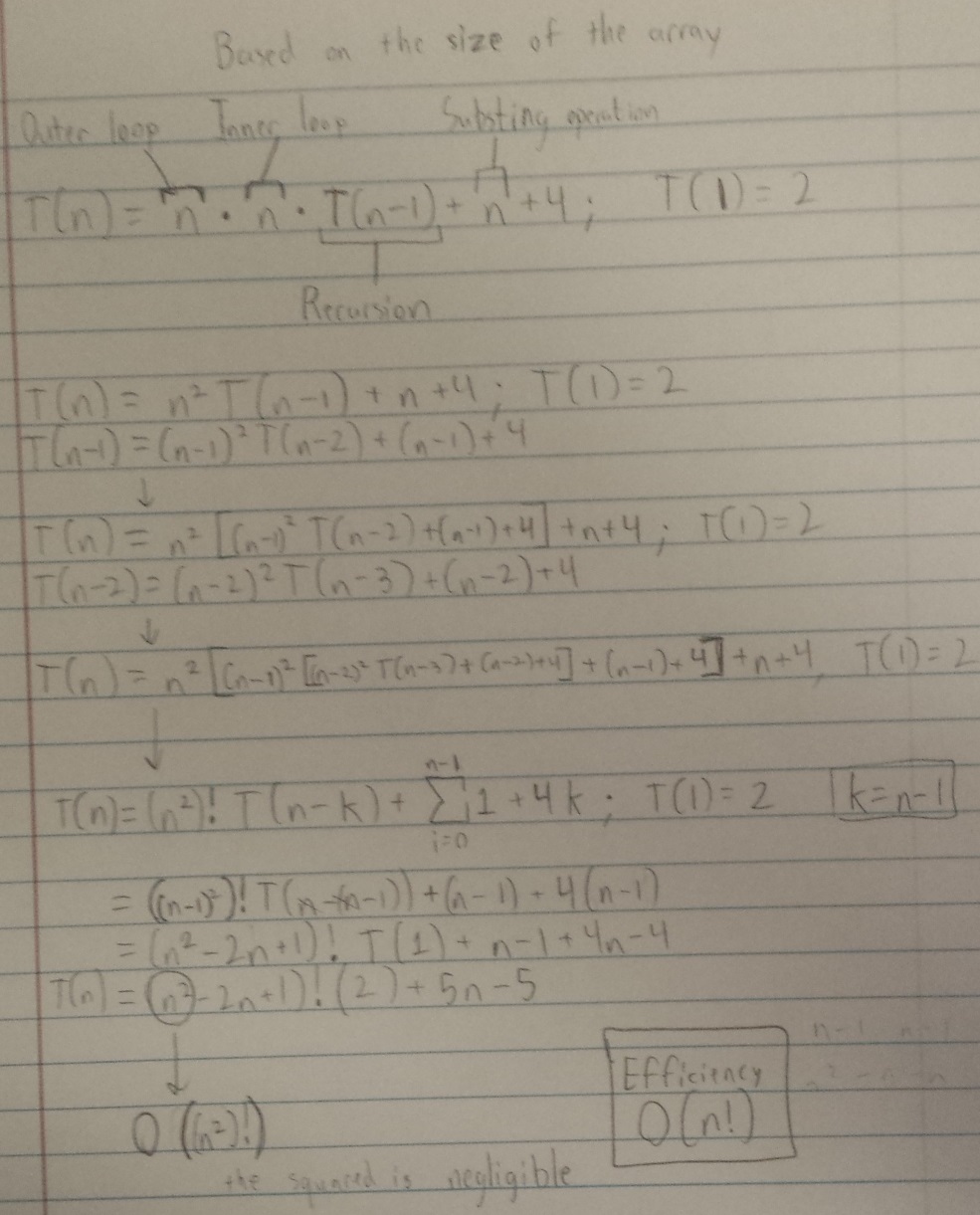
Algorithm:

* **ExhaustiveDecoder Class**
  + The algorithm begins by setting up a new ArrayList that will contain every valid, possible sentence the Morse code message can be. The ArrayList is of type MessageBigramResult, which is a nested class that holds a possible version of the Morse code message and the bigram frequency result of that message. Then the recursive method is called with two parameters, the Morse code string itself and a new ArrayList of Strings (decodedMessage) whose elements are the possible words that make up the decoded message.
  + In the recursive method, we first check the base case. The base case is whether the Morse code string is empty, meaning that we have searched each individual dot and dash in the message. If this is true, then that means that the decodedMessage array contains words and we can check its frequency result. If the result is greater than the threshold, then the algorithm saves the message and the frequency result into a new MessageBigramResult, which is then stored in the array list of possibilities.
  + If the base case fails however, then the algorithm goes to the next recursive case. Using the bigram frequency result, this case checks whether or not the current state of the decoded message is valid and if the algorithm should continue down this branch. If the frequency result is less than the threshold, then the algorithm will recognize this as a bad branch in the search and return out of the method to the last recursive call. If the result passes, then the algorithm will continue searching down the current branch.
  + After going through all those checks, the algorithm will go into a for-loop through all characters left in the Morse code message. In the loop, a substring of the Morse code from 0 to i+1 is stored. This substring of code is then used to get the set of all possible words associated with that combination of Morse code. If this set is not null, the algorithm then stores a substring of every Morse code character passed the ones used to get the words. A for-each loop is then created to loop through all of the possible words generate by the initial bit of Morse code stored at the beginning of the loop. In this loop, a new ArrayList of type String is initialized with the current decodedMessage array and given on of the words from the list of possible words. After all this, the recursive call is made by passing a smaller version of the Morse code and a larger list decoded words.
  + After the recursive method has gone through and gathered every possible version of the Morse code message, the recursive call returns to where it was originally called. Here, the array of possibilities is sorted from greatest to least bigram frequency using a quick sort algorithm, reduced to only the top 20 results in a new array, and then returned to the user.

Help from other students and classmates:

* Most of the help I received was focused on backtracking. For some reason, I have a hard time grasping the concept of backtracking and how it applies to certain situations. Because of this, I asked someone to help me understand backtracking and how it applied to this lab in terms of how we search through the Morse code message. Using what I learned, I was able to gradually develop and write the code I needed to effectively search through the Morse code message using backtracking.

Mathematical Analysis:



Empirical Analysis:

I included these tests because I wanted to test the result of my algorithm after having gone through to determine all possible versions of the Morse code. By increasing the length of the message and changing the content, I can see how the structure of the message affects the overall performance of the algorithm. Based on these test, I can conclude that the performance of this algorithm is dependent on more than just the length of the message and the content of that message. For example, there were two messages that were relatively the same in terms of length (34 and 38). However, the results were very different for the number of branches and completion time. This tells me that the algorithm’s performance is also dependent on the results of the bigram frequency tests the algorithm performs throughout the decoding process. These tests determine whether or not the algorithm should return and break off of the current branch. If the frequency of the words tested when decoding result in high frequencies, then the algorithm will continue down more and more branches, affecting the performance of the algorithm. In other words, having mostly common pairs of words in a message will lead to an algorithm that runs longer.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Message Length | Number of Possibilities | Number of Branching Paths  (One per return) | Message | Completion Time (in milliseconds) |
| 16 | 24 | 21706 | “WE ARE ONE” | 59 |
| 21 | 17 | 55007 | “WELL DONE” | 394 |
| 34 | 59 | 26044 | “I AM YOUR FATHER” | 233 |
| 53 | 183 | 396110 | “YOU CAN’T HANDLE THE TRUTH” | 802 |
| 38 | 57 | 229195 | “HE IS THE HERO WE NEED” | 1158 |